FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Converting LD to SFC (IEC 61131-3)

Vítor Emanuel Esteves Lopes

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Mário Jorge Rodrigues de Sousa

July 17, 2017

Abstract

Ladder Diagram is a simple graphic language useful to express control in terms of a set of boolean predicates that relate input boolean data to output actuation. Although widely used, LD is not properly designed to express sequential logic. To express sequential logic in a clear and structured way we can use SFC. SFC is a graphical language designed to break sequential control into small tasks and can be easily understood upon quick analysis. Since many industrial processes are sequential by nature and there are still many LD programs in the industry, a tool to extract sequential logic from LD programs and represent it in SFC is needed. From this premise, a software tool is developed in Java programming language. This tool takes a project in PLCopen XML format, extracts the LD program and creates the smaller state-space that represents the obfuscated sequential logic. From this state-space a SFC can be built.

Acknowledgment

To my father and to my mother, I have no words to describe how grateful I am for everything you've done for me. I have reached so far on your shoulders, and I'm sure I will achieve greater heights with your undeniable support. Thank you so much!

To Rita, the years we have spent together have been the best of my life. The countless years we shall spend together will be the best years of our lives. For all the love and support you've dedicated to me, I am eternally grateful. Also, thank you for all that blackboard and chalk used during my academic years - I'll be sure to get one for our home, so that I do not end up writting on the walls.

To my sister, who before me walked the academic path and from whom I learned countless things, who supported me without a break and was always there for me, thank you for everything.

To my brother-in-law, that one guy always available to lend me a hand and to cheer me up, thank you!

To my mother-in-law and to my sister-in-law, thank you for welcoming me into your family with open arms.

To my family, thank you for your support.

To my supervisor and professor, thank you for encouraging me to going further in this project, for all the support given during it and for all the things I learned.

To my friends, thank you for those incredible moments we spent together.

Vítor Lopes

"Software is a great combination between artistry and engineering. When you finally get done and get to appreciate what you have done it is like a part of yourself that you've put together."

Bill Gates

Contents

List of Figures

List of Tables

List of abbreviations and symbols

LD	Ladder Diagram
SFC	Sequential Function Chart
ST	Structured Text
IL	Instruction List
FB	Function Block
FBD	Function Block Diagram
XML	eXtended Markup Language

Chapter 1

Introduction

In 1968, one of the divisions of General Motors - the Hydramatic Division -, faced with the growing complexity and dificulty in maintenance and adaptation of the control systems to new production lines, gave start to a project for a system that could replace and improve industrial control systems. The control systems were mainly composed of electric circuits with switches, electromechanic devices for sequential control (cam timers, drum sequencers) and closed-loop controllers, kept in cabinets near the production lines. Bedford Associates took over the project and created the first PLC - the Modicon 084. In 1971, PLCs were starting to replace the old control systems. [6, 7]

The programming languages of the first PLCs were based on the electric circuits with switch logic, and were defined as "Relay Ladder Logic Diagram" or simply "Ladder Logic Diagram". The use of diagrams that resembled the old control techniques allowed for a better integration with engineers familiar to them. [6, 7]

With the growth of PLC technology, new companies emerged with their implementations and new programming languages. The differences in the way PLC were developed and programmed led to problems for the engineers that had to deal with PLC systems from different manufacturers. Due to the growing difficulty, the industry aligned into defining standards and IEC developed the IEC 61131 to standardise configuration and programming of PLC systems. [8]

IEC 61131-3, the third part of the IEC 61131 standard, defines 2 textual programming languages(IL, ST) and 3 graphical languages (LD, FBD, SFC). [1, 9]

Due to the way industrial automation developed over the years, Ladder Logic became one of the main programming languages for automation systems due to its origins. Ladder Diagram is a graphical programming language and can be defined as a interconnection of boolean variables in a ladder structure, where each rung defines a boolean expression. The resulting value of the expression is assigned to an output variable. It is mainly used for procesing boolean signals and its structure is not proper for use with sequential logic. [1, 9] SFC (Sequential Function Chart) is used to describe sequential logic with a proper graphical structure. It is composed of steps and transitions. The system starts from an initial step and goes from step to step by means of transitions. Each step represent a state of the system. Each step and transitions can be programmed

independently with any other languages, except SFC for transitions. [1, 9]

With the growth in usage of the standard, institutions like PLCopen [10] emerged. PLCopen works to promote efficiency in development and maintenance of IEC 61131 software, promoting the use of IEC 61131. Among various contributions, they developed PLCopen XML, an open standard to promote the interchangeability of programs between different IDEs. PLCopen XML is based in XML technology.

1.1 Motivation

Although Ladder Diagram is one of the main languages in the industry, its structure does not allow enginners who develop or maintain LD software to easily extract implicit sequential logic. To properly describe sequential logic there exists SFC. Its structure allows for a good understanding of the sequential nature of the system and is intuitive - one can understand the logic sequence by having a look at the SFC.

Therefore, the development of a tool that enables the extraction of implicit sequential logic and conversion of LD programs into SFC would benefit engineers who have to deal with these problems.

1.2 Objectives

The main objective of the project is to develop a software tool that allows the automatic conversion of LD programs with implicit sequential logic stored in PLCopen XML format to SFC programs. The output SFC must be a valid program, that can be edited in development environments that support PLCopen XML.

1.3 Structure of the document

The document is composed of 6 chapters, including this introduction.

In chapter 2 a study on the IEC 61131 standard, PLCopen XML technology and algorithms for conversion of LD to SFC available in the literature is made.

In chapter 3 analysis on the algorithms is provided and, based on the conclusions of the analysis, a solution is proposed.

Chapter 4 is dedicated to the software implementation of the proposed solution. In this chapter the software architecture, both high- and low-level, is presented.

In chapter 5 an example with a follow-through of the algorithms is provided. The results of the implementation are discussed.

Chapter 6 concludes the document and approaches the topics for future work.

Chapter 2

Literature review

2.1 The standard IEC 61131

When Programable Logic Controllers emerged, there was no standard way to program them. Each manufacturer would develop and sell its system independently of other manufacturers and provide the resources necessary to work with theirs. This included proper training in their proprietary programming languages and in the hardware architecture. PLC control systems were incompatible - each system required specialized personel to deal with it, and when there was a need to change to another system with different languages and architecture, there was the need to retrain engineers. This led to unnecessary inefficiency and high maintenance costs. [1, 9]

To address this problem, standard IEC 61131 was developed and published. This standard defines the hardware and software requirements a PLC system should fulfill to be compatible with other compliant PLC systems. [1, 9]

The standard defines itself as "guideline", meaning that it is expected that manufacturers implement only part of the requirements. To that end it provides a table with 62 features in which a manufacturer must point out those that their system complies to. [9]

IEC 61131 brings manufacturers together by unifying design approaches and enabling common investement and experience exchange. Manufacturers can work together to produce the tools they need, software developers can share the software modules they developed and manufacturers stand to gain from the reduced cost of buying tested, ready to deploy components developed elsewhere, i.e., if a manufacturer can acquire the small components it needs to build a complex system, then the focus is only on the integration of the components, reducing the time and cost spent on projects and allowing even more complex systems. [9]

For customers it means that training for each specific system is no longer needed, leading to cost reduction. It also means that customers can easily integrate/change components from different manufacturers. Although software exchanging between systems from different manufacturers is not guaranteed, the standard facilitates porting between them. [9]

IEC 61131 is divided in multiple parts. The first part (IEC 61131-1) defines basic terminology and concepts of a PLC system. The second part (IEC 61131-2) defines the hardware (eletronic

and mechanic) requirements and respective qualification tests. The third part (IEC 61131-3) defines the software requirements - the software structure of a PLC system and the programming languages supported. The fourth part (IEC 61131-4) provides guidance to PLC customers, by helping with choosing/installing/maintaining PLC systems. The fifth part (IEC 61131-5) concerns the communication between PLCs based on MAP Manufacturing Messaging Services. The sixth part (IEC 61131-6) concerns communication using Fieldbus technologies. The seventh part (IEC 61131-7) defines software elements for fuzzy logic support in PLC systems. The eighth part (IEC 61131-8) provides guidelines on implementing IEC 61131-3 programming languages. [1, 9]

2.2 IEC 61131, part 3

IEC 61131 part 3 (usually refered as IEC 61131-3) defines the software structures, including programming languages and its usage, and "Program Organisation Units" - the software blocks of which systems are composed of. [9]

Program development with IEC 61131-3 is well structured, allowing a program to be broken down in POU software blocks, with strong data typing, support for data structures and five languages to choose from. Each block can be programmed in one of the five languages. [1]

From the set of five languages, the standard defines two textual languages (IL and ST) and three graphical languages (FBD, LD and SFC). [1, 9]

Instruction List (IL) is a low-level language, similar to assembly, with a simple instruction set. Structured Text (ST) is a high-level language, similar to PASCAL, that offers the possibility of developing more complex programs with less instructions compared to IL. LD is a graphical language based on relay diagrams, where boolean logic is expressed with contacts and coils. FBD is a graphical language based on the interconnection of Function Blocks. SFC is a graphical language designed for breaking control tasks into steps. Each step executes a certain action. Steps can jump to other steps by means of transitions. [1, 9]

The work here described concerns only two of the five languages (LD and SFC), which are explored below.

2.2.1 Software architecture

2.2.1.1 Program Organisation Units

To structure the software, IEC 61131-3 defines software blocks called "Program Organisation Units" (POU). POUs are the smallest independent blocks in a IEC 61131-3 program. They are encapsulated and can be executed independently of other POUs. This behaviour allows for a high modularity and promotes the reuse of implemented and tested software. [9]

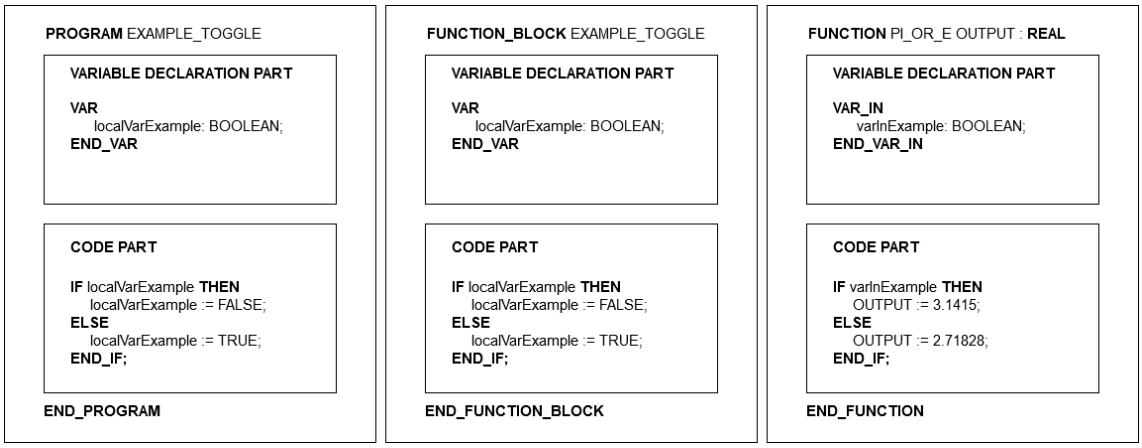


Figure 2.1: The three types of POU, as defined in the IEC 61131-3 standard.

A POU is composed of two main parts: variable declaration and code block. The header and footer define the POU type. [9]

Variable declaration The standard defines the usage of variables to store information. Each variable is declared with a certain data type and a certain variable type. Its types define how the variable is used, how it is accessed and what type of information they can store. The standard provides multiple datatypes and the possibility to create new ones derived from the elementary data types. [1, 9]

Bitstring types	Description
BOOL	Data type used to defined boolean values (1/0, TRUE/FALSE)
BYTE	8-bit bitstrings
WORD	16-bit bitstrings
DWORD	32-bit bitstrings
LWORD	64-bit bitstrings
Integer types	**Description**
SINT	Signed integer represented by 8 bits, allows integers in the interval [-128;+127]
INT	Signed integer represented by 16 bits, allows integers in the interval [-32768; +32767]
DINT	Signed integer represented by 32 bits, allows integers in the interval $[-2^{31}; 2^{31}-1]$
LINT	Signed integer represented by 64 bits
USINT	Unsigned version of SINT
UINT	Unsigned version of INT
UDINT	Unsigned version of DINT
ULINT	Unsigned version of LINT
Real types	**Description**
REAL	32-bit, floating-value, real number type
LREAL	REAL, but with 64-bit representation
Time & Date types	**Description**
TIME	TIME variables allow storing information about time intervals
DATE	DATE variables allow storing information about day, month and year
TIME_OF_DAY	To store absolute time values as marked on the clock
DATE_AND_TIME	Combination of DATE and TIME_OF_DAY
String types	**Description**
STRING	STRING types allow storing information of textual nature

Table 2.1: Data types defined by IEC 61131-3

It is also defined the use of generic data types. Generic data types are generalizations of the elementary data types and represent variables that can be of different data types. A generic data type starts with the prefix "ANY_". [1]

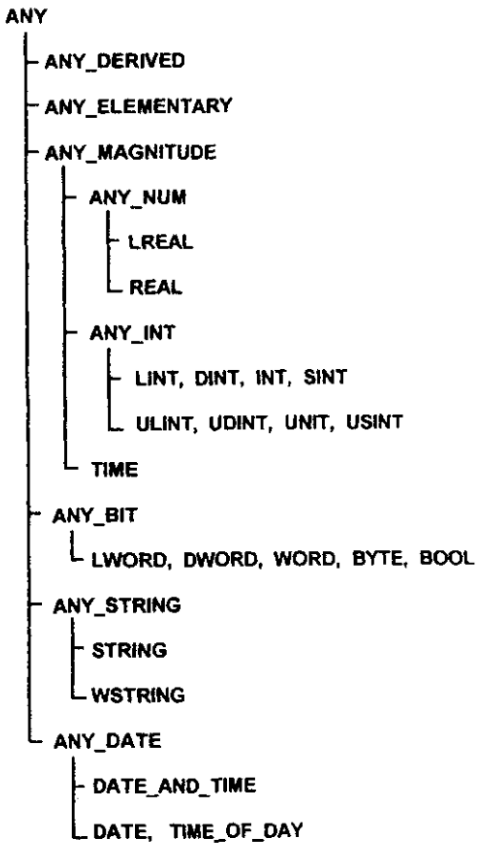


Figure 2.2: Hierachy of ANY_ usage. Taken from [1]

Variables must have a known initial value. By default, a variables' initial value is zero (null string in case of Strings, D#0001-01-01 for Dates). The initial value can be defined along with the data type in the declaration part. [1]

Listing 2.1: Defining initial value of a variable

```
exampleRealVar: REAL:= 3.1415;
exampleIntVar: SINT:= -5;
exampleBooleanVar: BOOL:= 1;
```

To create new data types, multiple ways are provided. New types are defined inside a TYPE / END_TYPE block. The name of the new type is define after the TYPE tag. Structured data types defines composite data types in a STRUCT block. Inside the STRUCT block, elements can be given a name and data type. [1]

Listing 2.2: Creating a new derived STRUCT data type

```
TYPE AQUARIUM
    STRUCT
        WATER_VOLUME : REAL;
        WATER_PRESSURE : REAL;
```

```
        WATER_TEMPERATURE : REAL;
        HAS_FISH : BOOL;
        NUMBER_OF_FISH : INT;
    END_STRUCT;
END_TYPE
```

Enumerated data types allow defining different names for different states of a value. These names are ordered in a enumerated list. [1]

Listing 2.3: Creating a new derived Enumerated data type

```
TYPE STATE:
    (INITIAL, PUMPING, DRAINING, CLOSING_VALVES,
    OPENING_VALVES, ERROR);
END_TYPE
```

Sub-range data types correspond to variables that have restriction in the range of values they can be assigned to. [1]

Listing 2.4: Creating a new derived Sub-range data type

```
TYPE
    WATER_VOLUME : REAL(0, +7.5);
END_TYPE
```

Finally, array data types allow the creation of arrays of values of multiple dimensions (although the standard doesn't define how many dimensions can be used). Arrays can be of any data type, even derived. [1]

Listing 2.5: Creating a new derived Array data type

```
TYPE AQUARIUM_COMPLEX_PRESSURES
    ARRAY[1..30] OF REAL;
END_TYPE
```

The variable type defines how the variable is accessed - it can be used as input parameter, output, or both. It can be local, internal within the POU or global. The table below sums up the different variable types. [1, 9]

Variable type	Description
VAR	Defines a local variable. Local variables can only be accessed in the POU where they are declared.
VAR_IN	Defines an input variable for the POU (interface). These variables are passed to the POU where they are defined as input parameters while being called by other POUs. The POU where the VAR_IN variable is defined can only read said variable.
VAR_OUT	Defines an output variable. The POU where VAR_OUT are declared can read and write on them. They are the output parameters of the POUs. Other POUs can read these variables, but not write.
VAR_IN_OUT	A combination of VAR_IN and VAR_OUT. Both the calling POU and the POU where the variable is declared can read and write. It behaves like pass-by-reference.
VAR_GLOBAL	This variable type defines global variables. They can be defined at CONFIGURATION level, RESOURCE level or PROGRAM level, depending on where they are declared. They are accessible to other POUs in the same CONFIGURATION / RESOURCE / PROGRAM.
VAR_EXTERNAL	If a POU needs to access a global variable (VAR_GLOBAL) then it must declare a VAR_EXTERNAL variable with the same name and data type to access the global variable.
VAR_ACCESS	Access variables allow the creation of access paths to input/output variables defined in a PROGRAM, global variables and directly represented variables. This allows remote devices to access these variables.
VAR_TEMP	VAR_TEMP are local variables that are put in a part of memory that is cleared after the invocation of the POU is finished.

Table 2.2: Variable types defined in IEC 61131-3

The standard also defines attributes to go along with the variable types. These attributes are: RETAIN, CONSTANT and AT. The RETAIN attribute indicates that the variables must be stored in non-volatile memory, so that the information is not lost when PLC loses power. CONSTANT attribute is used to indicate that the values of the variables cannot be modified. The attribute AT is used to define the memory location of a variable. [1]

The memory locations are organized into three regions: input memory, output memory and internal memory. Input memory stores data from analog and digital input modules. Output memory stores data to be sent to output channels. Internal memory allows storing intermediate values. These memory locations can be accessed directly without assigning a variable with the AT attribute. [1]

POU types There are three types of POU: PROGRAM, FUNCTION_BLOCK and FUNCTION. The type is declared on the header and footer of the POU. [1, 9]

FUNCTION FUNCTIONs are the most basic POUs. A FUNCTION POU has one or more input parameters in the form of VAR_IN or VAR_IN_OUT variables and returns one and only one output parameter. The variable and its data type of the output parameter must be defined in the header of the POU. [1, 9]

A POU of this type is not allowed to have memory - output is always function of the input parameters and the code inside. It cannot define or access global variables or define VAR_OUT variables. It is restricted to define and use only VAR_IN, VAR_IN_OUT or VAR, without the ability to use the RETAIN or AT attribute. [1, 9]

A FUNCTION cannot call FUNCTION BLOCKs or PROGRAMs, only other FUNCTIONs. They cannot be instantiated and they are executed whenever called. They must be available globally to the other POUs. [1, 9]

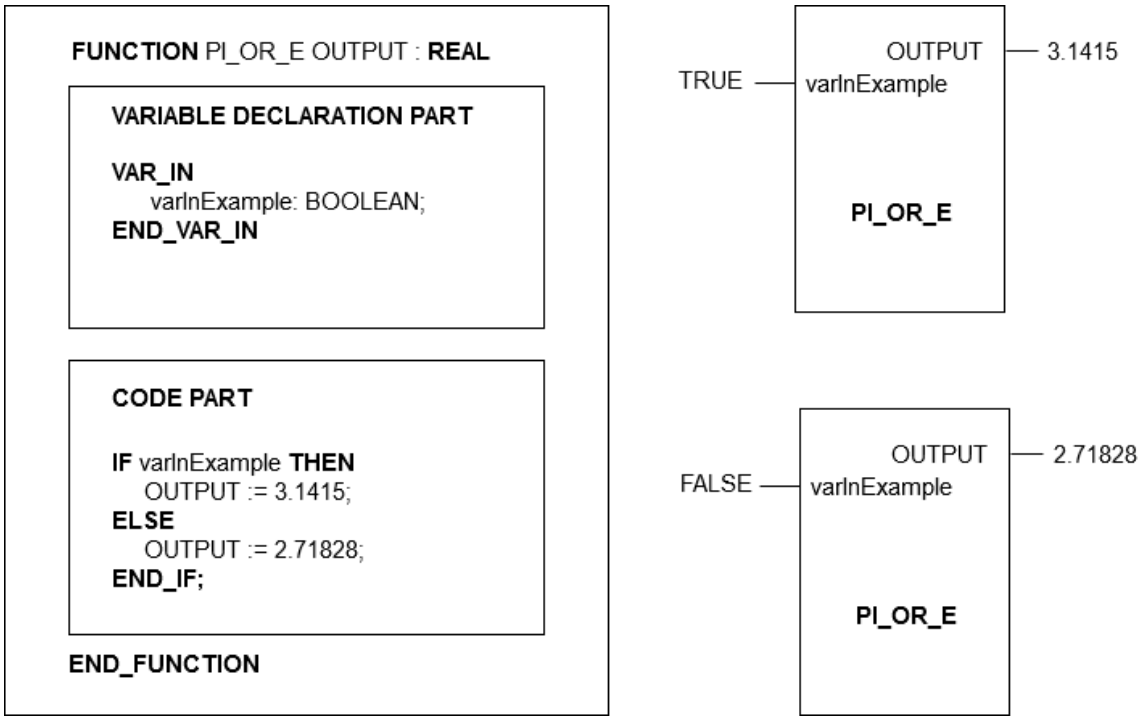


Figure 2.3: An example FUNCTION POU.

The EN input allows controlling the execution of the function. While set to FALSE, the function is not executed and a value is not assigned to the output. When set to TRUE, the function is executed and the output value is updated according to the input values. The ENO output is used to indicate if the function is properly executed. This facilitates the usage of FUNCTIONs in chains: the ENO output of one FUNCTION is used to set the EN input of the next FUNCTION. This guarantees that the next function is not executed in case of error. These variables are implicitly defined. [1, 9]

FUNCTIONs can only be defined in LD, ST, IL or FBD. [1, 9]

FUNCTION BLOCK FUNCTION BLOCKs are one of the main blocks to build programs, along with PROGRAM blocks. Like variables, FUNCTION BLOCKs must be instantiated in the variable declaration part of a POU. FBs can be instantiated and called in other FBs or in PROGRAMs. They can be declared as global variable or accessed with VAR_EXTERNAL. RETAIN attribute can be used on a FB instantiation to store them in non-volatile memory. Multiple instances of the same FB type can be instantiated in a FB or PROGRAM. [1, 9]

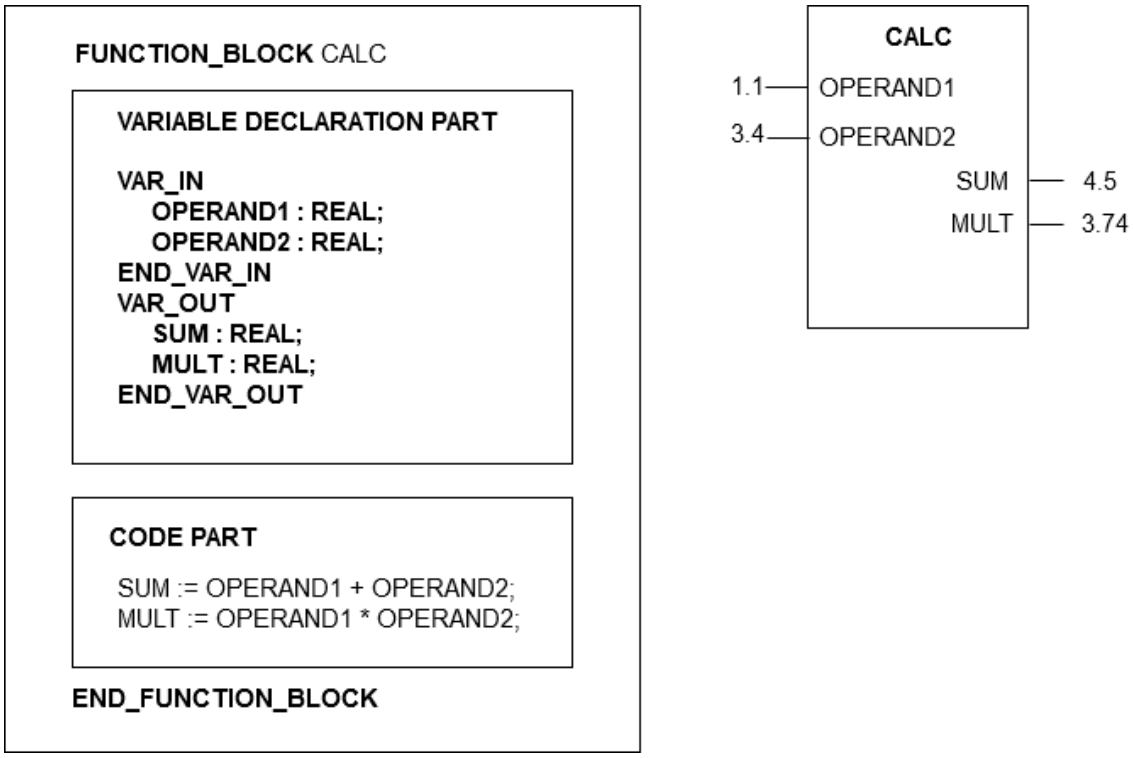


Figure 2.4: An example FUNCTION_BLOCK POU.

It is not possible to instantiate global variables in a FB, but global variables can be accessed via VAR_EXTERNAL. [1, 9]

FBs behaves like a data structure, instantiated in memory, with input/outputs accessible through the dot operator: <FBInstance>.input . [1, 9]

Data exchange with a FB is made through its input/output variables (interface). Each FB is encapsulated, independent of external events, and due to this, comparisons have been made to object oriented languages, although features like inheritance and polymorphism are not supported. [1]

FBs can be defined using all the five languages available in the standard. [1, 9]

PROGRAM These POUs are the what we could call the "main program" being executed in the PLC. They can instantiate global variables, FBs and FUNCTIONs and use VAR_ACCESS to

make variables available from remote devices. PROGRAMs are instantiated at the resource level and are only executed when associated with a "task". Like FBs, any language in IEC 61131-3 can be used to define it. [1, 9]

2.2.2 Ladder Diagram

Ladder Diagram is a graphical language based on relay diagrams, electric wiring diagrams used for industrial control before PLCs, familiar to engineers who worked with relay logic control schemes. It is easy to learn and understand, and, for small systems, it serves the industry well. [1]

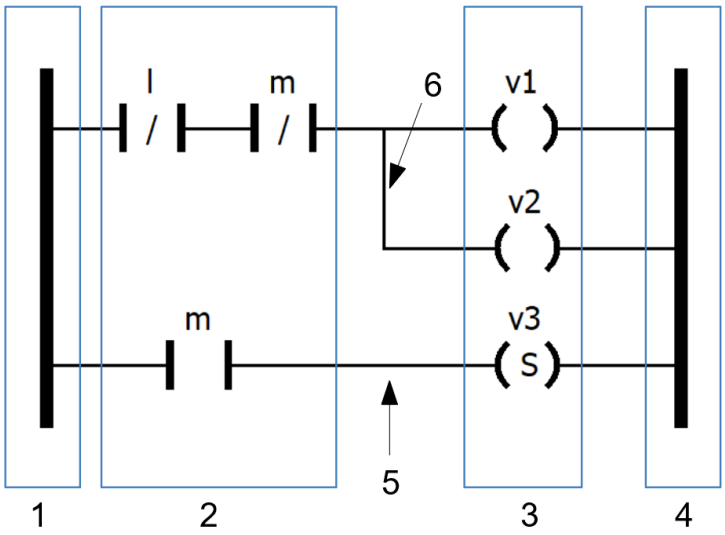
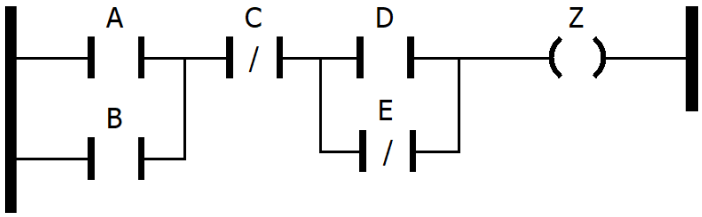


Figure 2.5: A simple Ladder Diagram example. 1 - Left Power Rail. 2 - Contacts. 3 - Coils. 4 - Right Power Rail. 5 - Horizontal Link. 6 - Vertical Link

A Ladder Diagram program is composed of two power rails, and one or more rungs connecting the left to the right power rail, although the right power rail is optional. The working principle is an adaptation of the working principle of electric wiring diagrams used for control: the left power rail provides power to each rung. The power flow in each rung is defined by the status and arrangement of the contacts in that same rung. At the right side, the rung ends with one or more coils. When power flow reaches the coils, they are energized. [1, 9]

Contacts and coils have only two states: ON or OFF (CLOSED or OPEN for contacts). Each element can be assigned one boolean variable. For contacts, the boolean variable status (true or false) defines if it is closed or open. For coils, the status of the coil (ON or OFF) defines the status of the variable (if it true or false). Through combinations of contacts, boolean expression can be defined. The boolean result is attributed to the coil. [1, 9]



$$Z := (A \vee B) \wedge (\neg C) \wedge (D \vee (\neg E))$$

Figure 2.6: A LD example and its respective boolean expression

The standard defines various types of contacts and coils. [1, 9, 8]

Contact type	Symbol	Description
Normally open contact	⊣ ⊢	The state on the left is copied to the right if the variable associated with the contact is TRUE.
Normally closed contact	⊣ / ⊢	The state on the left is copied to the right if the variable associated with the contact is FALSE.
Positive transition-sensing contact	⊣ P ⊢	If the variable associated with the contact becomes TRUE (positive transition), the state on the left is copied to the right for one evaluation cycle.
Negative transition-sensing contact	⊣ N ⊢	If the variable associated with the contact becomes FALSE (negative transition), the state on the left is copied to the right for one evaluation cycle.

Table 2.3: Contact types defined in IEC 61131-3. Symbols taken from Beremiz IDE editor [5]

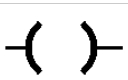
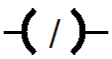
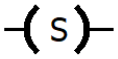
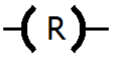
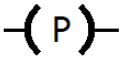
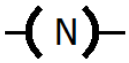
Coil type	Symbol	Description
Coil		The state on the left link (TRUE or FALSE) is copied to the associated variable and to the right link.
Negated coil		The state on the left link is copied to the right link. The inverse of the state on the left link is copied to the variable.
SET Coil		The variable associated with the coil is set - it becomes TRUE until reset.
RESET Coil		The variable associated with the coil is reset - it becomes FALSE until set.
Positive transition-sensing coil		The state on the left link is copied to the right link. The associated variable becomes TRUE for an evaluation cycle.
Negative transition-sensing coil		The state on the left link is copied to the right link. The associated variable becomes FALSE for an evaluation cycle.

Table 2.4: Coil types defined in IEC 61131-3. Symbols taken from Beremiz IDE editor [5]

Functions and function blocks In addition to these elements, the standard allows the usage of Functions or Function Blocks provided that inputs or outputs connected to rungs are of boolean type. [1, 9]

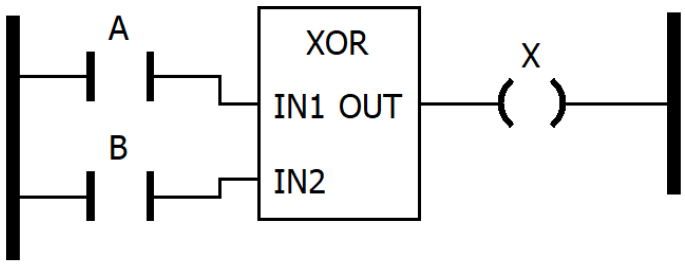


Figure 2.7: LD allows the use of Function Blocks in its rungs.

Jumps and labels Jumps and labels are also defined. These can be used for transfering control from one part of the Ladder Diagram to another. [1]

The IEC 61131 guidelines on using IEC 61131-3 recommend that these elements should not be used - it is not defined what should happen to the rungs following a rung with a jump element when that jump is active, although it is assumed they are not evaluated. [1]

Execution flow The evaluation of a Ladder Diagram, given that there are no jumps or labels to change the execution flow, is made from top to bottom, rung by rung. Each rung is evaluated from left to right. The standard defines rules to guarantee that evaluation is "unambiguous and consistent". [1]

- Input values must be available before being used in evaluation of an element. For example, before evaluating a coil, all contacts in the rung must be evaluated and their values must be available;
- The evaluation of an element is only complete if all the outputs of said element are properly evaluated - only then are the values of the outputs available for other elements;
- The evaluation of the complete network is only complete if all the outputs (coils) have been evaluated and have their values available;
- When transfering data from one network to another, all the values from the first network must be available before starting evaluation of the second network.

Feedback The standard allows the boolean variable assigned to a coil to be assigned to a contact in the same rung. This creates a feedback loop, where the value of the output of the rung is used as input. In rungs with feedback loop, the value used in the evaluation of the contacts is the value assigned to the coil in the previous evaluation. [1]

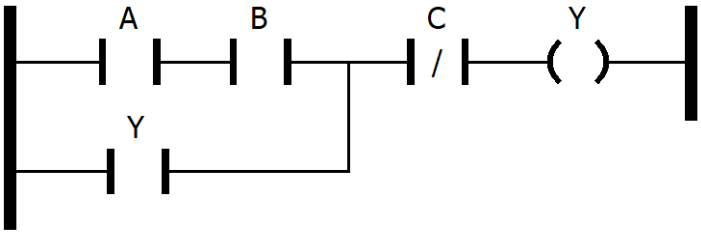


Figure 2.8: The variable Y is used both as coil and contact, i.e., the output has influence in its own state.

Advantages Due to its structure, LD is one of the best languages available in IEC 61131-3 to describe digital logic, mainly combinational logic. The low number of elements and the intuitive structure makes it simple, easy to understand and learn. It reached widespread usage due to its historical origins and it still is one of the main languages of industrial control. [1]

Disadvantages Although effective in describing digital logic, arithmetic operations and closed-loop control are not easily implemented with LD. And although it can be used to describe sequential logic, it's on the user to provide the necessary structures to properly describe sequential processes. Also, the fact that the application logic is mixed with sequence control makes the sequential behaviour hard to understand. [1]

Ladder Diagram does not lend itself well to large programs. As the program grows, the difficulty in creating well structured software grows. The support for software structures is low or non-existent, leading to difficulties in breaking complex programs in smaller parts and lack of true data encapsulation. [1]

[1] also points out the limitations in facilities for software re-use, the lack of support for handling data structures, the problem with evaluation of large programs that could lead to timing issues (larger programs take longer to evaluate, causing problems with time-critical software).

2.2.3 Sequential Function Chart

Sequential Function Chart (SFC) is one of the five languages, and one of the three graphical languages. SFC has origins on the graphical language Grafcet and, like Petri-net, it can be used to describe the behaviour of systems with multiple states. SFC is, therefore, the main language for sequential control available in IEC 61131-3. Due to its graphical nature, the possible states of a system and the transitions that could change are exposed and one can understand the sequential behaviour by having a look at the program. [1, 9]

SFC allows breaking a program in small steps, with each step having only the code needed for that step. If the program is not on a step, the code associated with that step is not executed. [1]

Although being mainly a graphical language, IEC 61131-3 also defines a textual form for SFC. [9]

Structure A SFC program is built with three elements:

- Steps
- Transitions
- Actions

A program is composed of multiple steps that represent the state the system is in. Steps and transitions are interleaved: after a step there must be a transition, and after a transition there must be a step. A transition represents a condition that, when true, triggers the deactivation of the previous step and the activation of the next step. [1, 9]

Steps There are two types of step: initial step and normal step. An initial step must be unique in a SFC - it represents the starting point of the program. Each step has two variables associated with it: [1]

- A flag of BOOLEAN type that stores the state of the step: if it is active the flag is true, else it is false. It can be acessed in the program with the form <StepName>.X;
- A variable of type TIME that stores the information about the time passed since the last activation. Each time the step is activated, this variable is reset. It can be accessed through <StepName>.T.

Transitions Each transition has a boolean condition. While the condition evaluates to false, there is no transition firing. Whenever the evaluation of this condition returns true, the transition fires. This means that the previous steps are deactivated and the following steps are activated. [1, 9]

The condition can be defined in ST, IL, LD or FBD and the result must always be of BOOLEAN type. [1]

Transitions can lead to multiple steps, through divergent paths or simultaneous paths. The standard defines "divergent", "convergent", "simultaneous sequence divergence" and "simultaneous sequence convergence". [1]

Simultaneous paths correspond to sequence of steps that can be active at the same time. Divergent transitions correspond to choosing between various possible sequences. Divergente paths can converge onto a step with convergent transitions. [1]

Actions Each action corresponds to an instruction sequence that is executed when the associated step is active. The code is executed in loop while the step is active. This code can be written in any of the other languages in IEC 61131-3. [1]

The action name must be unique in the program and can be called in more than one step. [1]

An action has a qualifier that defines how the action must be executed. Calling actions is not restricted to SFCs. They can also be called in LDs and FBDs, although it is not recommended. [1]

The different qualifiers defined in the standard can be seen in the next table.

Without qualifier	In the absence of qualifier, it is assumed to be N
N	The action executes while the step is active
S	The qualifier S makes the action start execution on the associated step and only stop execution when the action is reset, even if the SFC transitions to another step.
R	Qualifier R makes the action stop being executed, i.e., it resets the execution of the action.
L $TIME$	Imposes a temporal limit on the execution of the action. Action starts execution when steps becomes active and stops execution after the defined time interval TIME passes or if the step is deactivated.
D $TIME$	Delayed action. The action only executes after the time interval TIME if the step is still active. If the step is deactivated before, the action is not executed. The action stops being executed when the step becomes inactive.
P	With P qualifier, the action is only executed once, at the moment of step activation.
P1	Similar to P, the action is executed once at activation. The use of P1 instead of P is recommended.
P0	With P0, the action is executed once at deactivation of the step.
SD $TIME$	"Stored and delayed" - like D qualifier, the execution of the action is delayed, but with SD, the action keeps being executed after the step is deactivated. It is a combination of qualifiers S and D.
DS $TIME$	"Delayed and stored" - differs from SD in the fact that the action is not executed if the associated step becomes inactive before the time interval TIME.
SL $TIME$	"Stored and Time Limited" - the actions starts execution when the step becomes active and is executed during time interval TIME even if the step becomes inactive.

Table 2.5: Actions qualifiers as defined in IEC 61131-3

Textual SFC The standard defines the possibility to describe a SFC graph in textual form. Four constructs are provided:

Listing 2.6: The four constructs for building textual SFC

```
INITIAL_STEP "name":
    "action"
END_INITIAL_STEP
```

```
STEP "name":
    "action"
END_STEP
```

```
TRANSITION FROM {"setOfStateNames"} TO {"setOfStateNames"}:
    "boolean condition"
END_TRANSITION
```

```
ACTION "name":
    "action code"
END_ACTION
```

2.3 PLCopen XML

2.3.1 PLCopen

"PLCopen" is an organisation based in the Netherlands, with offices in USA, Japan and China [10], that aims to be a group with common interest in the acceptance of Industrial Control standards, in the promotion of compatibility between PLC systems and development of tools to lower the costs and rise eficiency in software development. [9, 10]

The members of PLCopen are entities with interest in the wide acceptance of the use of standards in the field of Industrial Control and range from manufacturers and supplier to educational institutions. [10] Members commit to produce or employ IEC 61131-3 compliant systems. [9]

As a result, PLCopen provides reusable and hardware-independent software for motion control with the motion control library of function blocks [11], defines safety related aspects through software [12], worked with OPC Foundation to provide the open, hardware and software independent communication protocol OPC-UA, based on OPC technology [13] and developed the standard format PLCopen XML, based on XML, to ease the exchange of project information between compliant systems [14]. For every solution provided, PLCopen also provides certification and certified training. [15]

2.3.2 Beremiz

Beremiz is a free, open IDE compliant with IEC 61131-3 and other open standards. It aims to provide engineers the possibility to develop their work independently of the brand of PLC, freeing them of vendor locks and in a interoperable-rich environment. [5]

The IDE offers an editor for IEC 61131-3 languages compliant with PLCopen XML. This means that programs are stored in XML format, and can be open in any platform that supports the PLCopen XML standard. [16]

2.3.3 XML & XML Schema

XML (eXtensible Markup Language) is defined as a tool to store and transfer data between sytems. It is a human-readable markup language, meaning that the information is organized with tags, arranged hierarchicaly. In XML, information is written in plain-text, making it platform-independent. This aspect makes it a good tool to exchange data between systems without compatiblity problems. [17]

Listing 2.7: Small example of a XML structure, taken from [17]

```
<note>
    <to>Receiver</to>
    <from>Sender</from>
    <heading>Message header</heading>
    <body>Message contents</body>
```

```
</note>
```

The example provided is a simple case of information structuring with XML. The information about a message is organized with tags. The tag "<note>" and its respective ending tag "</note>" enclose the information about the note, which has multiple fields. These fields are represented by the tags enclosed inside the <note> and </note> tags. These tags - <to>, <from>, <heading> and <body> are children of the tag <note>. In this case, the tag <to> stores information about the receiver, <from> about the sender, <heading> the title of the message and <body> the content of the message.

Tags are "elements" and can possess "attributes". An element is considered everything from the starting to the ending tag, including attributes. It is possible to define empty elements, without information inside. [17]

Listing 2.8: Expanded example with attributes and empty elements.

```
<note priority="high">
    <to>Receiver</to>
    <from>Sender</from>
    <heading encoding="UTF-8">Message header</heading>
    <body encoding="UTF-8">Message contents</body>
    <footer type="signature"/>
</note>
```

The attributes can be seen inside the opening tags, like the "priority" attribute in <note> or the "enconding" attribute in the <heading> and <body> elements. The footer does not have parseable data inside, so it can be a self closing element, an empty element. These attributes define information about the tags they are in. Attributes values must be always quoted. [17]

Nesting tags When nesting tags - tags inside a tag - the child tag must be closed before its parent is closed. [17]

Listing 2.9: Wrong way of nesting. This results in errors. <to> must be closed after being opened, before note is closed.

```
<note>
    <to> Someone
    </note>
</to>
```

Listing 2.10: Proper nesting.

```
<note>
    <to> Someone
    </to>
</note>
```

Tree structure XML has a tree-like structure. There must be a root element that encloses all other tags. [17] In the case of the previous example, <note></note> is the root element.

There is no pre-defined set of XML tags. The tags must be defined by the user. To standardize a group of tags and its attributes, along with the hierarchy between them, DTD or a XML schema can be used. [17]

DTD and XML schema To define how a XML document must be structured for a certain application, DTD and XML schemas are the proper tools. They not only define structure, but they can also be used to validate XML documents that must comply with the defined structure. With DTD and XML schemas, standards on how to interchange data with XML can be built. [17]

Listing 2.11: DTD file for the example in listing 2.7, taken from [17]

```
<!DOCTYPE note
[
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
]>
```

DTD are plain-text files that define the legal elements in a XML. The "!DOCTYPE" defines the root element. The other elements are defined in a list enclosed in square brackets under the root definition. To define an element, "!ELEMENT" declarations are used. Elements with children elements have those elements' names described between brackets after the that element declaration. "#PCDATA" is used to describe elements that have parseable text data. [17]

DTD also permits the declaration of "!ENTITY" types. These entities are used to define special characters or strings with keywords that can be used in the XML to refer to the same characters or strings. [17]

Listing 2.12: XML schema file for the example in listing 2.12, taken from [17]

```
<xs:element name="note">

<xs:complexType>
  <xs:sequence>
    <xs:element name="to" type="xs:string"/>
    <xs:element name="from" type="xs:string"/>
    <xs:element name="heading" type="xs:string"/>
    <xs:element name="body" type="xs:string"/>
  </xs:sequence>
</xs:complexType>

</xs:element>
```

XML schemas provide structure definition written in XML. XSD files (XML Schema Definition) define elements and its attributes, the data types and default values. The <xs:element> tags define elements. <xs:complexType> are used to define elements that have children elements. <xs:sequence> is used to define that the complex type element is a sequence of other elements. [17]

2.3.4 PLCopen XML standard

PLCopen XML [18] defines a standard for data exchange based on XML. This standard not only provides the ability to store and transfer textual information, but also graphical information, such as position and size of graphic elements used in the languages of IEC 61331-3. It is intended that transfer of programs between development environments can be done without much effort from the user, without loss of information.

Since PLCopen XML documents stores the data of a IEC 61131-3 project, they are not only useful for exchanging information, but also for simulation and modeling tools, for verification, documentation and version control. [18]

To define the structure and the legal elements and attributes, PLCopen XML uses XML Schema Definition (XSD). To that end, it provides the XSD file where these elements are described.

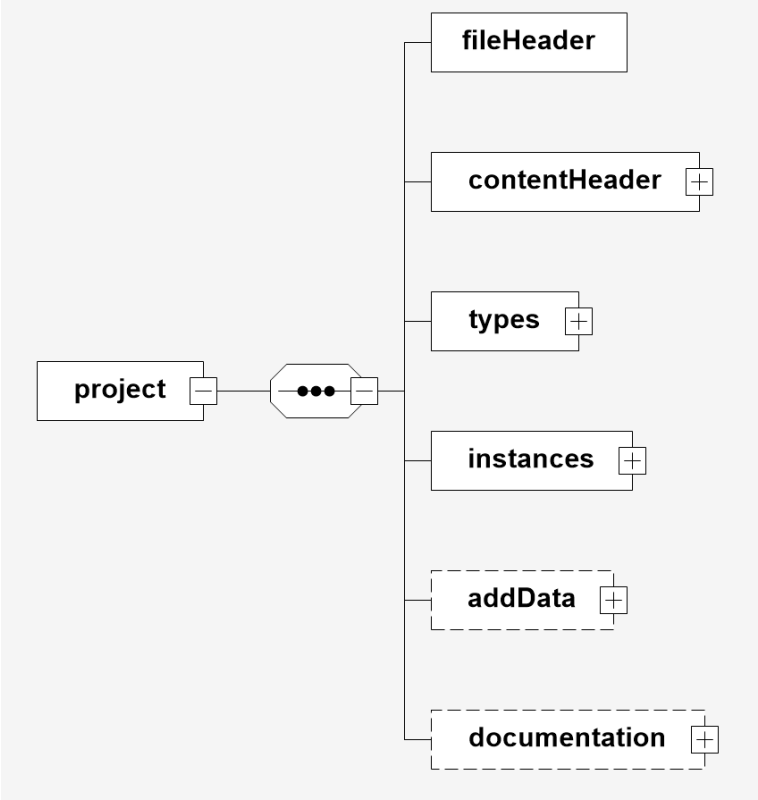


Figure 2.9: The root element "project" and its child elements.

The root element is the "project" element. The "project" element is a complex type, with a sequence of elements.

Headers The "fileHeader" element provides "information concerning the creation of the export/import file". [18] It has a total of 7 attributes, 4 of which are obligatory, and no child elements. The attributes are:

- "companyName" - required
- "companyURL" - optional
- "productName" - required
- "productVersion" - required
- "productRelease" - optional
- "creationDateTime" - required
- "contentDescription" - optional

Information about the company, the project and the creation date must always be supplied. An example produced with Beremiz can be seen in the listing 2.13.

The "contentHeader" element stores information about the project. It has 6 attributes, with only 1 obligatory - the name of the project. [18]

- "name" - required
- "version" - optional
- "modificationDateTime" - optional
- "organization" - optional
- "author" - optional
- "language" - optional

It also has one obligatory child element "coordinateInfo", which stores information about the coordinate system, as defined in the PLCopen XML standard. [18] An example is provided in listing 2.13.

Listing 2.13: The "fileHeader" and "contentHeader" of a project developed in Beremiz, stored in PLCopen XML format

```
<fileHeader companyName="CompanyExample"
productName="DrillMachineProduct"
productVersion="1"
creationDateTime="2017-04-10T11:04:17"/>

<contentHeader name="DrillMachineProject"
modificationDateTime="2017-06-17T11:50:57"
author="Vitor">
    <coordinateInfo>
        <fbd>
            <scaling x="0" y="0"/>
        </fbd>
        <ld>
            <scaling x="0" y="0"/>
        </ld>
        <sfc>
            <scaling x="0" y="0"/>
        </sfc>
    </coordinateInfo>
</contentHeader>
```

Types The "types" element stores information about datatypes and POUs in the project. When a user defines a new data type based on elementary, derived or extended data types, it is defined under the child element "dataTypes" as a "dataType" element. The POUs defined in the project are stored under the child element "pous" as "pou" elements. These two child elements have no attributes. [18]

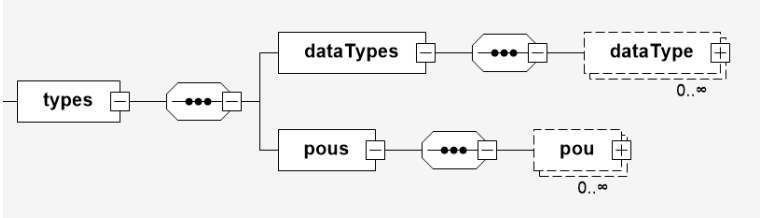


Figure 2.10: The element "types" stores information about data types and POUs in the project

Data types A "dataType" element has an obligatory attribute called "name", which identifies the user-created data type. [18]

A Beremiz example is provided in the listing 2.14.

Listing 2.14: The information about a user-created data type

```
<dataTypes>
  <dataType name="ExampleDataType">
    <baseType>
      <struct>
        <variable name="ElementIntExample">
          <type>
            <DINT/>
          </type>
        </variable>
        <variable name="ElementBoolExample">
          <type>
            <BOOL/>
          </type>
        </variable>
        <variable name="ElementRealExample">
          <type>
            <REAL/>
          </type>
        </variable>
      </struct>
    </baseType>
```

```
    </dataType>
  </dataTypes>
```

We can see that the user-created data type is a derived base type struct with three variables: ElementIntExample of type DINT, ElementBoolExample of type BOOL and ElementRealExample of type REAL.

A "dataType" element has the obligatory "baseType" child element which defines the base type for the defined data type. It can be an elementary base type, a derived base type (struct, array, etc) or extended base type (pointer). [18]

A list of base types is provided in PLCopen XML standard. The following elementary types are defined [18]:

BOOL, BYTE, WORD, DWORD, LWORD, SINT, INT, DINT, LINT, USINT, UINT, UDINT, ULINT, REAL, LREAL, TIME, DATE, DT, TOD, STRING, WSTRING.

The following derived types are defined [18] :

ARRAY, DERIVED, ENUM, SUBRANGESIGNED, SUBRANGEUNSIGNED, STRUCT.

And the extended type: POINTER.[18]

Generic types are also defined [18] :

ANY, ANY_DERIVED, ANY_ELEMENTARY, ANY_MAGNITUDE, ANY_NUM, ANY_REAL, ANY_INT, ANY_BIT, ANY_STRING, ANY_DATE.

POUs A "pou" element stores information about a POU in the project. A "pou" has 3 attributes, from which 2 are obligatory. [18]

- "name" - required
- pouType - required
- "globalId" - optional

The "pouType" can be one of three types: "function", "functionBlock" and "program". [18]

A "pou" has 4 main types of child elements: "interface", "actions", "transitions" and "body". A "pou" can have various "body" elements. The possibility to define various bodies allows to break the POU in smaller sections, called "worksheets". [18]

A body has two attributes: "WorksheetName" and "globalId", both optional. A "body" element encloses the elements that define the code part of the POU. The "interface" encloses the elements that define the declaration part of the POU. [18]

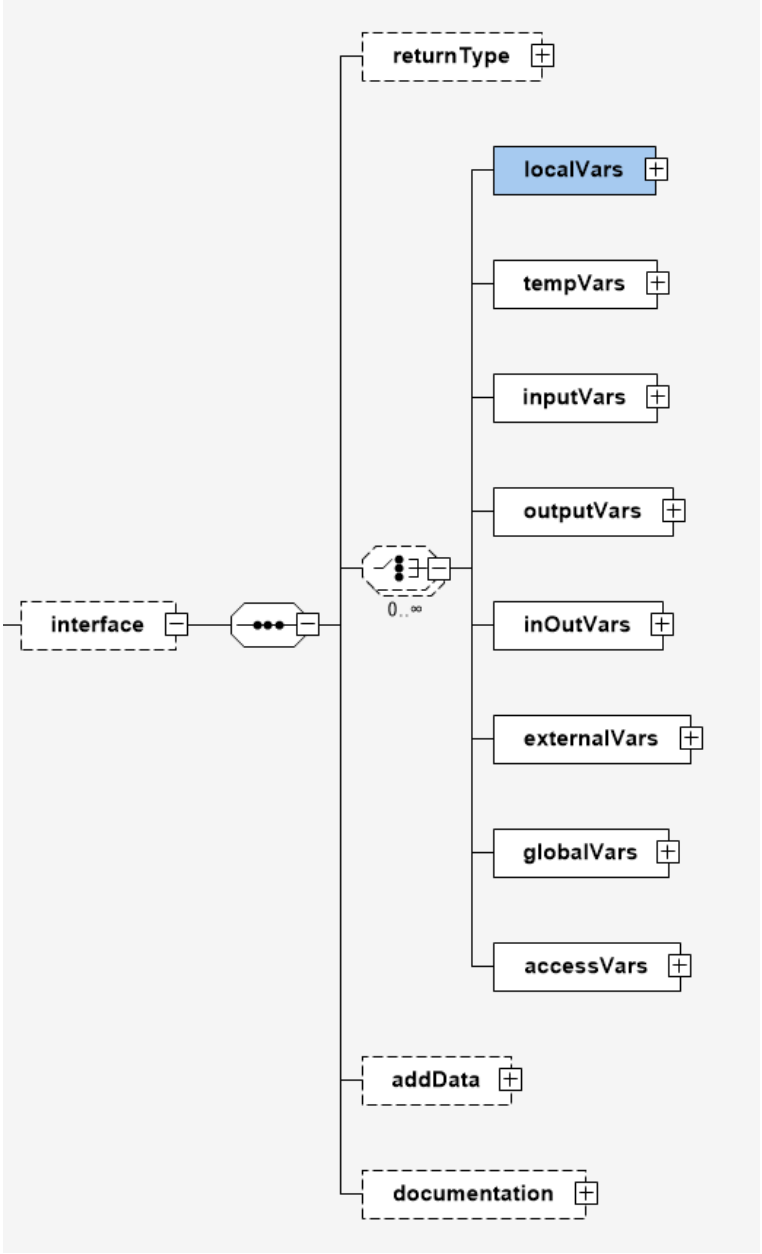


Figure 2.11: The element "interface" is present in any POU and stores information about its declaration part.

The interface can define a return type, which is one of the data types available (elementary, derived or extended). This return type applies to functions' return variable. Local, temporary, input, output, input/output, external, global and access variables related to the POU can be defined here. [18]

Each element (localVars, ...) has zero or more elements of type "variable". [18]

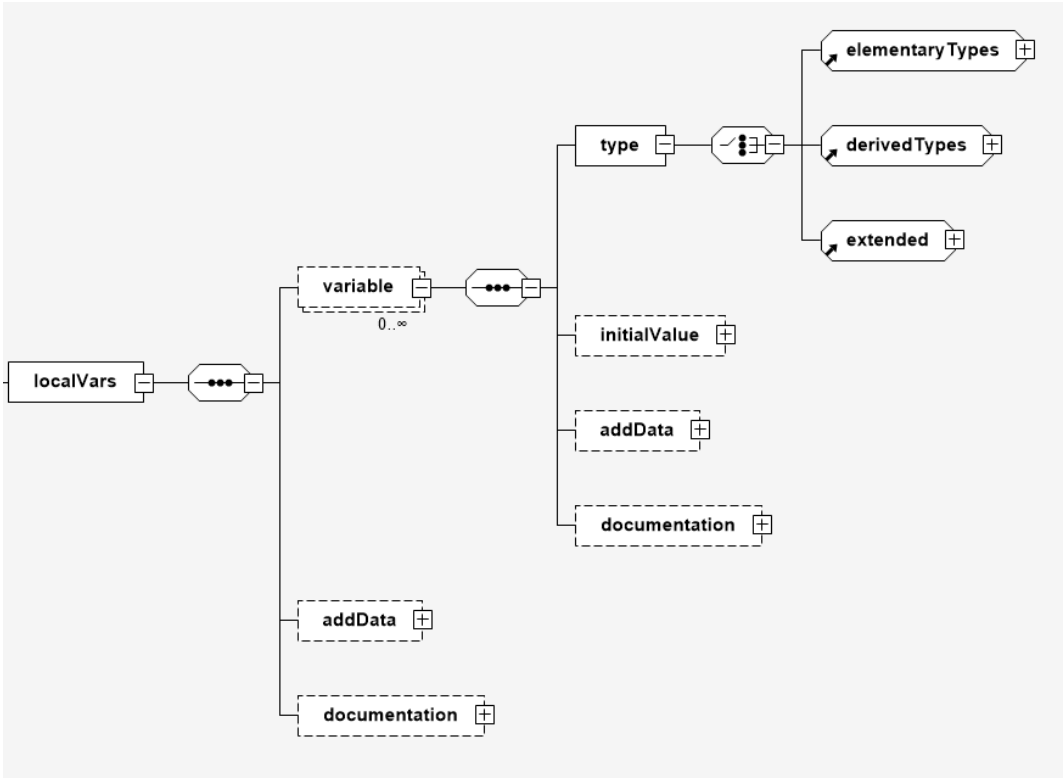


Figure 2.12: The element variable defines a variable in a IEC61131-3 project.

The element "variable" has a required attribute called "name", which identifies the variable. A "variable" has two main child elements - the type of the variable (which is one of the elementary, derived or extended types) and an initial value, as optional element. [18]

The "body" element encloses elements that define the five languages. To define ST or IL code, normal text is used. To define LD, FBD or SFC code, graphical elements are provided. These graphic elements have information about their position in the coordinate system, the connections between objects, the variable elements associated with the graphical element where applied and other information. [18]

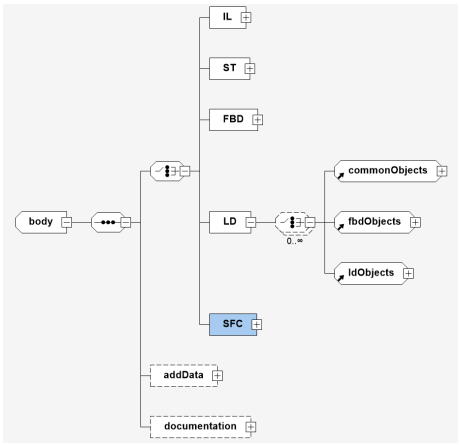


Figure 2.13: A body of a POU can be described in any of the five languages in IEC 61131-3.

There are 4 types of objects: "commonObjects", "fbdObjects", "ldObjects", "sfcObjects". [18]

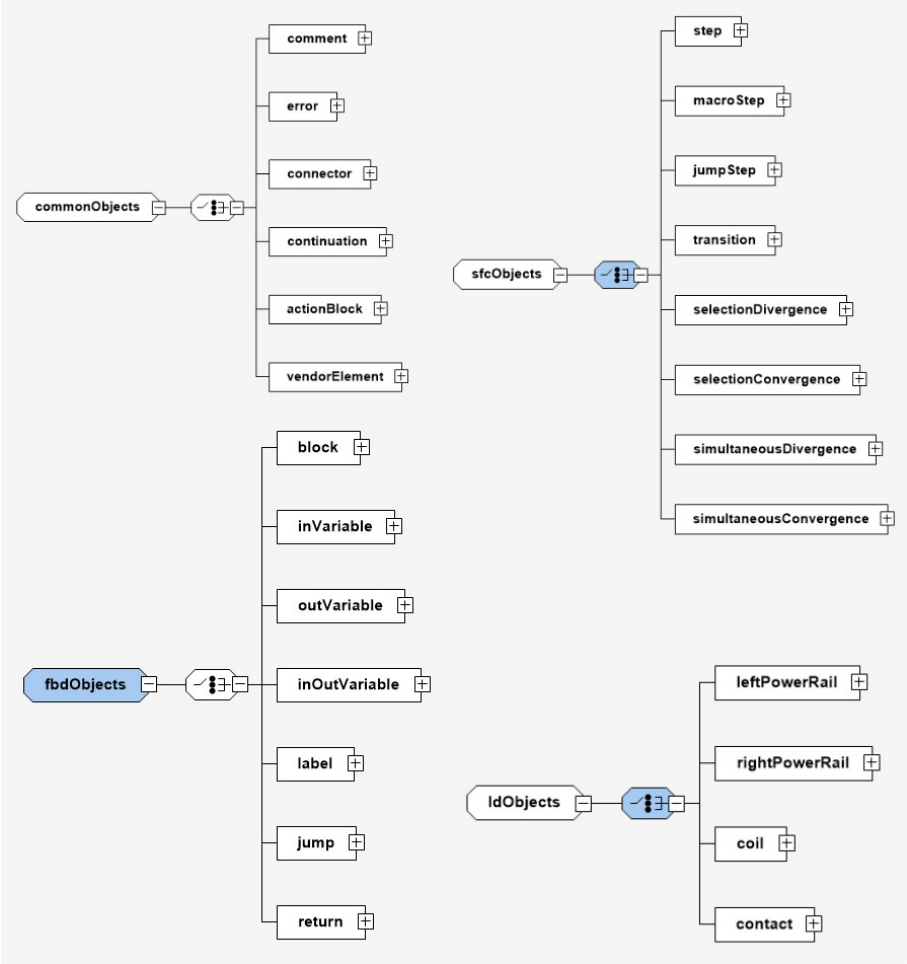


Figure 2.14: The set of graphical objects defined in PLCopen XML

Instances The "instances" element, defined under the "project" element, stores information about the configurations of the project. This includes resources, tasks and other information about configuration. [18]

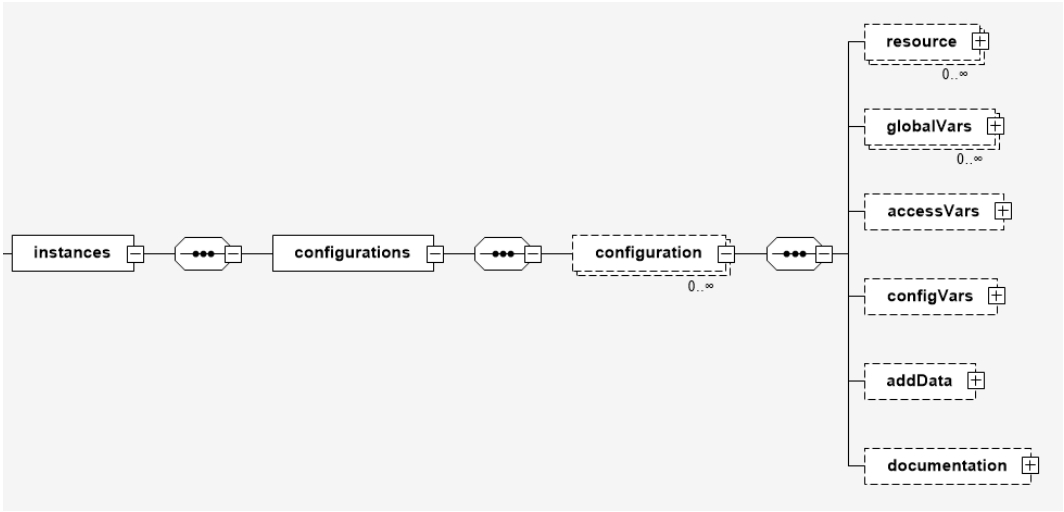


Figure 2.15: The "instances" element and its children

2.3.5 XML DOM

XML DOM (Document Object Model) defines a way to access and manipulate XML documents. It is a W3C standard and defines a programming interface for XML documents i.e., defines "how to get, change, add or delete XML elements". [17]

DOM deals with XML documents by considering each element a node and building a node tree to represent the document. DOM defines properties and methods to deal with every node, i.e. every element. [17]

2.4 Algorithms in literature to convert LD into SFC

This section deals with conversion algorithms found in the literature. These algorithms focus on extracting the sequential logic hidden in the LD and represent it in SFC.

The first algorithm [3] is based on a graphical approach - graphs and graph decomposition methods are the tools used to build SFC from a Ladder Diagram.

The second algorithm [2] approaches the problem considering that information from sequential logic cannot be extracted from the LD alone and that a plant model is needed. To model the plant, the authors present temporal logic methods. These models represent the information that the first algorithm [3] ignores.

The third is based on state-space extraction. The author considers that the events in the plant are asynchronous. From the extracted state-space, heuristic rules are used to extract feasible states. These rules convey knowledge about the plant. From the feasible states, sequence is extracted.

2.4.1 The RLL Design Recovery Algorithm

The Design Recovery algorithm by A. Falcione and B. Krogh [3], published in 1993, is the first proposed algorithm to convert a Ladder Diagram to a Sequential Function Chart by extracting the implicit sequential logic. The algorithm targets relay logic with boolean elements, with the objective of building a SFC that emulates the behaviour of the LD. It supports the construction of SFC with both sequential and parallel paths.

As it stands, the algorithm has no formal mathematical basis and its complexity is unknown. The authors acknowledge this and point out that these topics must be improved.

The algorithm makes no use of user knowledge about the plant being controlled by the LD. It also assumes that state variables are initialized to zero (false) and that each state variable is the output of only one rung. State variables correspond to the outputs that change as function of input variables, the outputs of rungs.

The algorithm is based on the partitioning of the Ladder Diagram in groups of smaller rungs which are then organized in a sequential graph depending on the relationship of precedence and parallelism between them, i.e., groups of rungs that cannot be active at the same time are organized along single-path structures and groups that can are placed in parallel paths in relation to each other. The rungs are assigned to SFC steps. When a step becomes active, the rungs associated with it are scanned.

2.4.1.1 Graphs

To make the partitioning, an iterative procedure based on graph decomposition is described. The authors define three types of graphs [3]:

- Simultaneity Graph;
- Dependency Graph;

- Condensed Simultaneity Graph.

A simultaneity graph stores information about state variables that can be active at the same time in the LD. This graph has as many nodes as there are state variables in the LD, with each state variable assigned to a node. Situations where state variables can be active at the same time are defined by edges connecting the nodes of those variables.

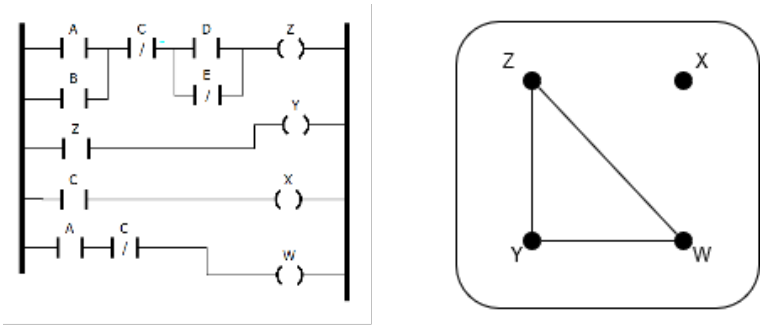


Figure 2.16: Simultaneity graph

A dependency graph, in the same way as a simultaneity graph, has one node for each state variable. This graph stores information about the dependencies of state variables. If the activation of a state variable in the LD depends on other state variable in a previous rung then an arc A(i, j), with i != j, connects node i to node j. This means that the variable assigned to node i depends on the variable assigned to node j.

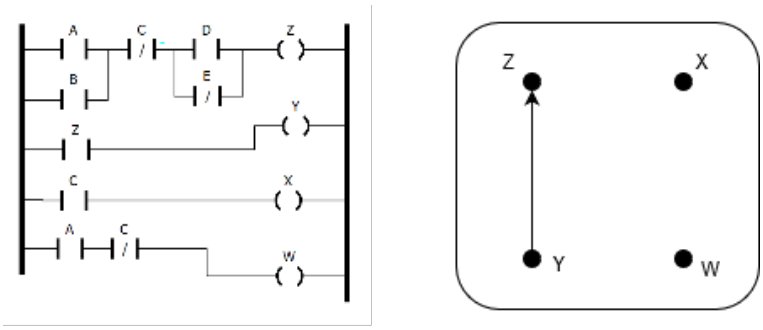


Figure 2.17: Dependency graph

A condensed simultaneity graph merges the previous two graphs in a single one. To that end, nodes that are connected in the dependency graph are grouped in a single node in the condensed simultaneity graph. The connections in the simultaneity graph are carried to the new graph.

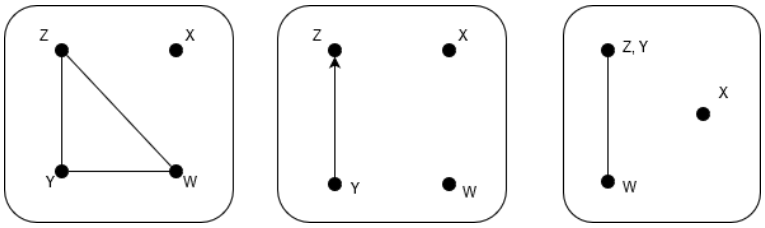


Figure 2.18: Condensed Simultaneity graph as the junction of the two graphs on the left

2.4.1.2 Graph decompositions

The authors also define two types of graph decompositions [3]:

- Connected Component Decomposition (CCD)
- Full Connectivity Decomposition (FCD)

The decompositions are operations upon a graph that leads to the sub-division of that graph into multiple graphs.

The CCD consists in dividing a graph into its connected nodes - groups of nodes that are connected are kept in the same graph, leading to isolated graphs.

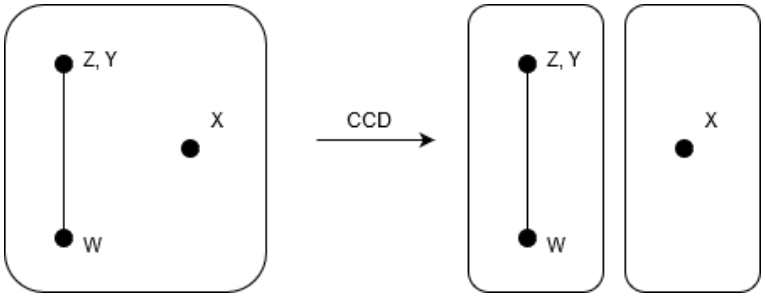


Figure 2.19: CCD procedure on the condensed simultaneity graph of figure 2.18

The FCD consists on the division of a graph in a way that nodes that are connected to every other node in the graph get their own graph.

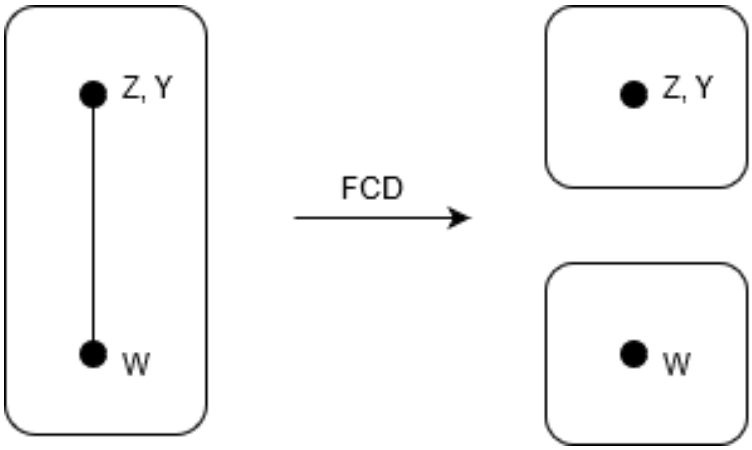


Figure 2.20: FCD procedure on the condensed simultaneity graph obtained in figure 2.19

2.4.1.3 The algorithm

There are five steps: two inital steps, two iterable steps and a step to verify the end of the algorithm.

The first step comprises the building of the inital SFC. The initial SFC is composed of two steps: an initial step with no actions and a normal step to which the Ladder Diagram is assigned. The initial step transitions into this normal step and this normal step transitions back to the initial step. Whenever any of the rungs is to become active, the SFC must transition to the normal step. This means that the condition to activate this step is the OR of all the logic expressions defined by the contacts in each rung. When there are no rungs active, the SFC must transition back to the initial step. In this case, the condition is the AND of the negation of the state variables, which is the AND of the negation of the logic expressions that activate the state variables. This SFC is the starting point to the algorithm. [3]

The second step consists in the building of the graphs. First the simultaneity graph and the dependency graph are built. Then, with both graphs, the condensed simultaneity graph is created. This graph is assigned to the step in SFC to which the LD was assigned. [3]

The third step is one of the two iterable steps. For every normal step in the SFC, there is a condensed simultaneity graph associated. Perform CCD in each graph. Where CCD can be performed, two or more sub-graphs will be created from the original one. The step where CCD was performed is replaced by a sequence of steps, with as many steps as the number of sub-graphs created. Each sub-graph is associated to a step in that sequence. The order of the steps in that sequence must be found. For that, a new graph is introduced: sequence graph. [3]

A sequence graph [3] is a graph composed of as many nodes as the number of sub-graphs produced in the CCD. Arcs A(i, j) connect nodes in the graph. An arc from node i to node j means that the step j comes after step i in the sequence. To determine these arcs, activation and deactivations conditions for each step must be found. Let activation condition of step i be A_step_i and the deactivation condition for the same step i be D_step_i.

A_{step_i} is given by the OR of the rung conditions associated with step i, tanking into account that state variables are considered logic-0. D_{step_i} is given by the AND of the NOT of the state variables associated with step i.

An arc(i,j) is placed iff (D_{step_i} -> A_{step_j} OR A_{step_j} -> D_{step_i}) AND NOT (D_{step_j} -> A_{step_i}) OR A_{step_i} -> D_{step_j}), i != j . [3]

After defining all the arcs, unnecessary paths are removed. A path from node i to node j is removed if there is another path from node i to node j that includes the set of nodes traversed by the former.

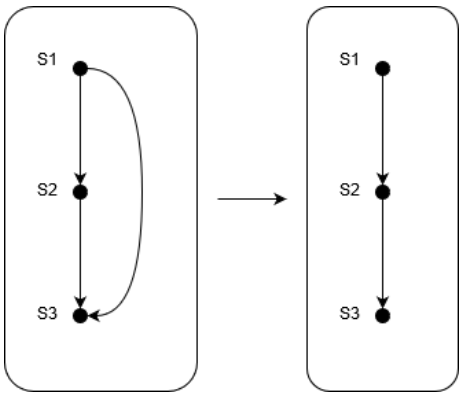


Figure 2.21: Eliminating an unnecessary path

With sequence order defined, transition conditions are determined. [3]

If step i is the first step in the sequence, the condition for the transition that leads to step i is the OR of the activation conditions (A_{step}) of the steps in the sequence reachable from that transition.

If the transition is the deactivation transition of a terminal step i in the sequence or the transition between step i and step j such that it is the only path from step i to step j, the condition is the deactivation condition of step i D_{step_i}.

If the transition connects step i to step j, but it is not only transition from step i (there are other steps k_n reachable from step i) then transition condition is given by the AND of the deactivation condition of step i D_{step_i} with the activation conditions of steps k_n $A_{step_k}n$.

At last, a dummy step is added before the sequences that replace the original step. This dummy step leads to all first steps in the sequence, and the terminal steps in the sequence lead to the dummy step. The dummy step also leads to the rest of the SFC.

The fourth step is the second iterable step in the algorithm. For every graph in the current SFC, perform the FCD on it. If FCD can be performed, two or more sub-graphs will be created. For each sub-graph, create a parallel path. To each parallel path a sub-graph is assigned to the step in that path. [3]

The fifth step states the end condition of the algorithm: repeat step 3 and step 4 until CCD and FCD cannot be performed in every graph in the current SFC; when graphs can no longer be decomposed, the conversion is complete. [3]

2.4.2 Extraction of action order with temporal logic models

In [2], Zanma et al. provides analysis on Falcione & Krogh algorithm[3], stating that "the information about parallelism and precedence of events cannot be extracted from the LD alone". They consider that most of the SFCs produced by Falcione & Krogh algorithm are, therefore, "useless". They present an algorithm based on temporal logic that takes into account the closed-loop between Ladder Diagram and the controlled plant.

2.4.2.1 Temporal logic

Temporal logic provides the tools to describe temporal precedence relationships between events in a system. To that end, it adds new operators to the set of operators in classic logic.

The authors focus on three operators.[2] Let σ represent a sequence of events, let p represent an event and j represent a discrete time belonging to the set of non-negative natural numbers. The formal definition of the occurence of an event p, part of a sequence of events σ, at discrete time j is given by:

$$(\sigma, j) \models p \tag{2.1}$$

The formal definition of an event p, part of a sequence of events σ, occuring in the next discrete time is given by:

$$(\sigma, j) \models \bigcirc p \quad iff \quad (\sigma, j+1) \models p \tag{2.2}$$

The formal definition of an event p, part of a sequence of events σ, eventually occurs is given by:

$$(\sigma, j) \models \diamond p \quad iff \quad (\sigma, k) \models p \quad para \quad k \geq j \tag{2.3}$$

2.4.2.2 Sensors, actions, timers and memory

In [2], LD variables are organized within the following sets:

.list M_s set represents starting switches; X set contains plant sensors; Y set contains the actuators; T set contains the timers used in the LD; M set contains the variables stored in memory.

2.4.2.3 Plant modeling

The authors make use of temporal logic to model the plant and transform the classic logic in the LD into temporal logic. A plant is modeled as a set of event precedence relationships, defined as P. [2] This set defines the dynamic behaviour of the plant. Consider the following example:

$$P = \{Y_1 \Rightarrow \diamond X_1, Y_2 \Rightarrow \bigcirc X_2, X_1 \Rightarrow \neg X_2, X_2 \Rightarrow \neg X_1\} \tag{2.4}$$

The $Y_1 \Rightarrow \diamond X_1$ implication means that, if Y1 is true, then eventually X1 becomes true. This type of implication is good to model cases where actions lead to non-instantaneous changes in the plant. The $Y_2 \Rightarrow \bigcirc X_2$ implication means that, if Y2 is true, the next discrete time X2 will be true. This implication helps to model scenarios where the event occurs instantaneously after the action, scenarios where the sensor data resultant of the event is available the next evaluation cycle. Along with temporal implications, normal implications can also be used.

Plant model is essential to the success of the algorithm. If the model lacks information or there are logical errors, wrong assumptions are made and the resulting SFC does not match the expected result.

In [2], Zanma et al. defines other sets of interest. One is the initial state set Θ. Its elements are the variables with boolean true value when the system starts. It is defined as

$$\Theta = \{x | x \in \{X \cup Y \cup T \cup M\} \quad e \quad (\sigma, 0) \models x\} \tag{2.5}$$

The authors also define the Y_P set.

$$Y_P = \{x | x \in \{Y \cup T\} \quad e \quad x \quad aparece \quad em \quad P\} \tag{2.6}$$

The Y_P is the set of actions and timers that appear in the set P, the model of the plant.

The sets $A(j)$, $B(j)$ and $B^A(j)$ are also defined.

$$A(j) = \{x | x \in X \quad e \quad (\sigma, j) \models x\} \tag{2.7}$$

$$B(j) = \{x | x \in \{Y \cup T \cup M\} \quad e \quad (\sigma, j) \models x\} \tag{2.8}$$

$$B^A(j) = \{x | x \in \{Y \cup T\} \quad e \quad (\sigma, j) \models x\} \tag{2.9}$$

$A(j)$ groups the sensors which are active in discrete time j. $B(j)$ groups the output variables which are active in discrete time j. $B^A(j)$ is a smaller set of $B(j$, since $B^A(j)$ groups only the output variables who have direct impact in the plant. Therefore, internal variables are not considered in the $B^A(j)$ set.

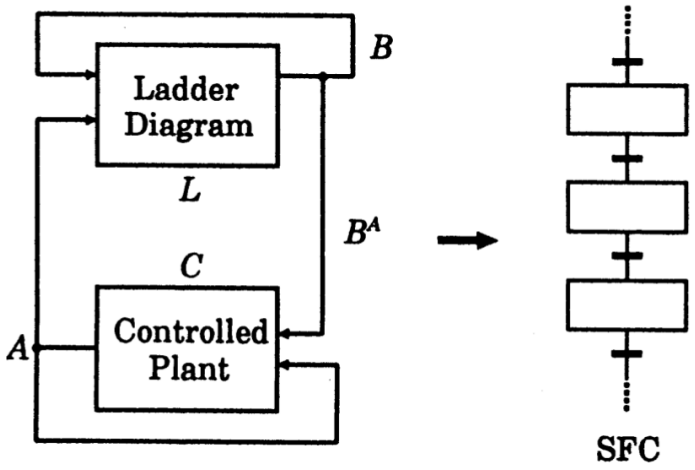


Figure 2.22: System representation in closed loop. Taken from [2]

The state of the plant the next discrete time $(A(j+1)$ is dependent on the actions executed in the plant $(B^A(j+1))$ and current state $A(j)$. That dependence is defined by the model and is represented by function C. C is determined from the plant model P. [2]

$$C(A(j),B^A(j+1)) = A(j+1) \tag{2.10}$$

2.4.2.4 About the order of evaluation of the LD

The next state of the outputs of the system (LD + controlled plant), given by B(j+1), is defined by the current state A(j) and B(j) and by the transformation introduced by the LD.

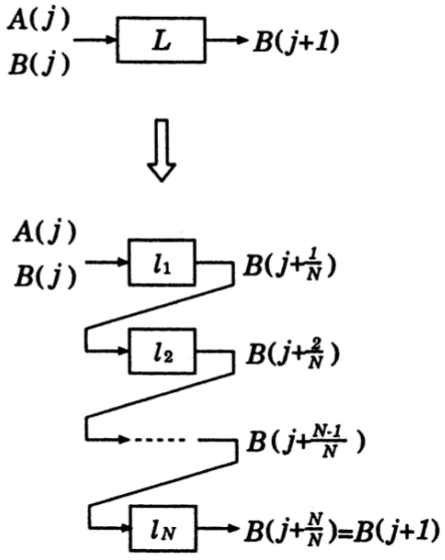


Figure 2.23: Obtaining the next state from the current state and the LD program. Taken from [2]

While evaluating the Ladder Diagram to obtain $B(j+1)$, the order of evaluation (from top to bottom) must be taken into account. A change in the output of a rung can lead to a change in other rungs ahead. Falcione & Krogh [3] dealt with these rungs by lumping them into single nodes, as previously seen. With Zanma et al., the next state $B(j+1)$ is determined rung by rung. This is reflected in the figure 2.23 as $B(j+1/N)$, $B(j+2/N)$, ... , $B(j+N/N)$. $B(j+1)$ is only properly defined when all the rungs are scanned. [2]

2.4.2.5 The remaining sets

Three more sets are defined. [2]

$$A_{\diamond}(j) = \{x | x \in X \quad e \quad (\sigma, j) \models \diamond x\} \tag{2.11}$$

The set $A_{\diamond}(j)$ defines the sensors that, at discrete time j, eventually will become active. The authors do not define how this set can be built, but is is necessary knowledge about the plant to know when certain sensors might become active. The only source of knowledge about the plant is the plant model.

$$\varepsilon(i)(j) = \{x | x \in X \quad e \quad x \in A(i) e \quad x \notin A(j)\} \tag{2.12}$$

$$\varepsilon^{B}(i)(j) = \{x | x \in \{Y \cup T \cup M\} \quad e \quad x \in B(i) e \quad x \notin B(j)\} \tag{2.13}$$

These two aggregations define variables which are active at discrete time i, but inactive at a subsequent discrete time j. Like the previous set, the authors do not define how to build these sets, although knowledge about the plant is implied.

2.4.2.6 Conditions for parallel paths

Situations where parallel paths might occur in the SFC are verified by testing two conditions [2]:

$$\text{Condition 1:} \quad |B^{A}(j) \cap \{Y_P \cup T\}| = 1 \tag{2.14}$$

$$\text{Condition 2:} \quad |B^{A}(j+1) \cap \{Y_P \cup T\}| \geq 2 \tag{2.15}$$

If both conditions are true, parallel divergence starts at j+1.

2.4.2.7 The algorithm

The first steps of the algorithm comprise the creation of the plant model and the definition of the initial sets (M_S, X, Y, T, M, Y_P, Θ).

The algorithm consists in repetition of some steps until an ending condition is reached.

First step: obtain $B(j+1)$ and $B^A(j+1)$ by scanning the LD with the current state $B(j)$ and $A(j)$. Verify if the conditions of parallel divergence are true. If parallel divergence conditions are met, go to the third step. Else, go to the second step. Stop the algorithm if $A(j) \cup B(j) = \Theta, (j >= 1)$.

Second step: determine the values of X that change by means of the function C, i.e., determine how the inputs change $(Aj+1)$ when $B(j+1)$ determined before affects the plant. If $A(j+1) = A(j)$ and $B(j+1) = B(j)$, modify $A(j+1)$ so that $A(j+1) = A(j) \cup A_{\diamond}(j)$. Go back to the first step.

Third step: Take each output belonging to the set $B^A(j+1)$ and create a SFC step for each one, all in a parallel divergence. Then apply the first and second steps to each sequence in parallel, without considering the influences of "other sensors". When no new elements are obtained in $B^A(j+1)$ for each sequence, i.e., when $B^A(j+1) = B^A(j)$, a waiting step is assigned and marks the end of that sequence. Do the same for every sequence in parallel. At that point all sequences have a waiting step. Then, $A(b_e)$ and $B(b_e)$ are determined, with b_e the discrete time after the waiting steps - the synchronization step.

$$A(b_e) = \bigcap_{i=1}^{n} \{A(b) - \varepsilon^A(b, b+f_i)\} \cup \bigcup_{i=1}^{n} \{A(b+f_i) - (A(b) - \varepsilon^A(b, b+f_i))\} \tag{2.16}$$

$$B(b_e) = \bigcap_{i=1}^{n} \{B(b) - \varepsilon^B(b, b+f_i)\} \cup \bigcup_{i=1}^{n} \{B(b+f_i) - (B(b) - \varepsilon^B(b, b+f_i))\} \tag{2.17}$$

According to [2], these equations can be understood as follows: "if an element is changed in the parallel divergence, then its changed value is used, while if there is no change of an element, then its value is that before the parallel divergence began".

2.4.3 Extraction of implicit sequential logic with state-space based approach

The third algorithm, by Nandhan & Babu [19], is a state-space based, computing-driven approach. It consists on producing a state-space by toggling inputs, one at a time, to produce a list of states that is trimmed down by heuristic rules. These rules define certain combinations of variables, and therefore states, as unfeasible. The heuristic rules convey user knowledge necessary to remove impossible states from the state-space. From the list of feasible states, a sequence is extracted.

Nandhan & Babu [19] refer the work of Falcione & Krogh [3], stating that the algorithm is not mature to deal with pratical cases. They also refer [2], stating that developing temporal models and the arbitrary selection of initial state "hamper the use of this approach for any pratical problem" [19].

The algorithm can be broken in three parts [19]:

- State-space generation;

- Feasible states extraction;
- Sequential logic recovery.

2.4.3.1 State-space generation

Initially, the boolean equations described by the LD are used to find the initial state: all inputs are set to zero and the values for the outputs are found by evaluating the LD.

A state is formed by the inputs and outputs and respective boolean values. The first state has all inputs logic-0 and outputs with the boolean values found on LD evaluation.

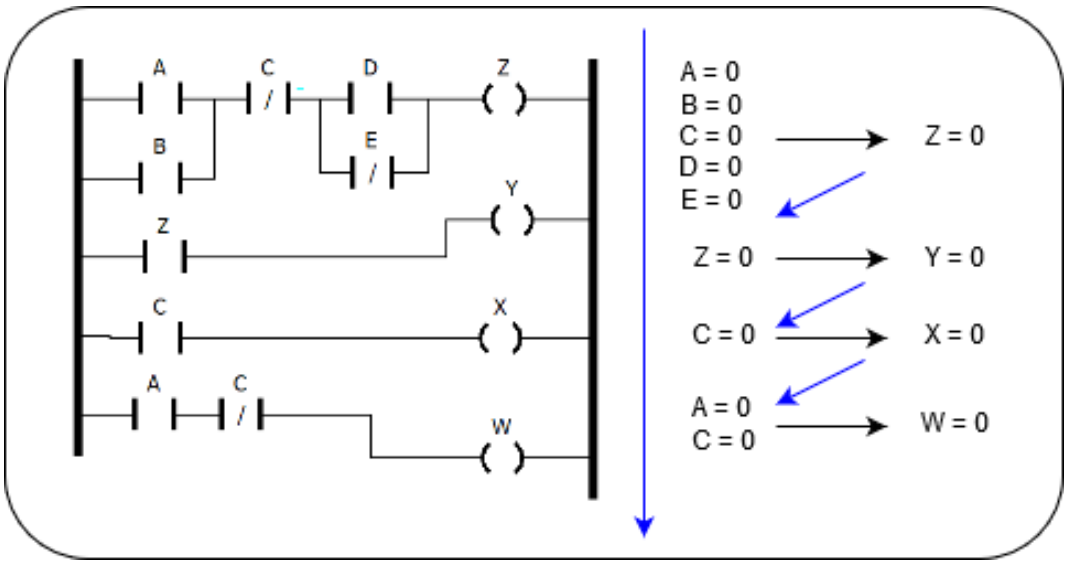


Figure 2.24: The LD is evaluated from top to bottom. The initial state is determined by setting all input logic-0.

In the previous example, the first state is given by: A=0, B=0, C=0, D=0, E=0, Z=0, Y=0, X=0, W=0.

The first state is added to the state-space. Then, the algorithm starts its iterative process to find all states: take the next state in the state-space and toggle only the first input. Evaluate the LD with the new input set and get the output set for that input combination. This forms a new state. If that state isn't already in the state-space, it is added to it. Then toggle only the second input, and so on [19].

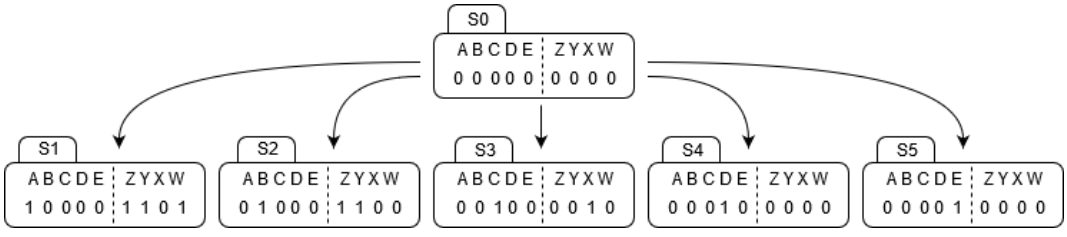


Figure 2.25: From the next available state (in this case, the first state) five more are produced by toggling each input.

The same process is repeated with every new state. When no new states can be produced, the algorithm reaches the end of state-space generation.

2.4.3.2 Feasible states extraction

The second part of the algorithm is the extraction of the feasible states. These states are those that comply with a set of heuristic rules defined by the user. These rules convey information about the plant and define that certain combination of variables is mandatory, for example, "if the sensor X is ON, then sensor Y must be ON also" or "sensor A and sensor B are mutually exclusive". States where these rules are not obeyed are deemed unfeasible state and are removed from the state space. [19]

Considering the previous example (figure 2.25), if the heuristic rules define that D must have the same value as E, states S4 and S5 are removed because they are unfeasible according to the rules defined.

2.4.3.3 Sequential logic recovery

With a reduced set of states (due to removal of unfeasible states), the third part of the algorithm starts. In this part, outputs are analysed. For each output, the list of states is traversed. In states where that output is active, the inputs that do not change value among those states are identified as "necessary input condition" for that output. The process is repeated for all outputs. [19]

When every "necessary input conditions" [19] are found, the sequential logic extraction begins. The authors propose an heuristic method that works as follows: the output with the least "necessary input conditions" is the first output to be actuated, the one with the most "necessary input conditions" is the last to be actuated. Outputs with the same set of "necessary input conditions" are actuated simultaneously. If two outputs, O_1 and O_2, have different sets of "necessary input conditions", but in those sets there are elements common to both, and the common elements form a set that activates another output O_x, then O_1 and O_2 are in parallel path following the activation of O_x.

Chapter 3

Analysis on the literature and solution proposal

This section deals with analysis on the three algorithms described in the literature review. From the conclusions taken, a solution is proposed. A planning for the project is described.

3.1 Context and problem identification

Although it has heavy use in the industry, Ladder Diagram is a simple language which structure does not allow easy extraction of implicit sequential logic. Engineers who deal with Ladder Diagram program would benefit from having the implicit sequential logic described in a proper sequential structure. SFC is a graphic language developed to describe sequential behaviour. It is intuitive and eases the understanding of the sequential nature of industrial processes.

Therefore, the development of a software tool for the conversion of LD programs into SFC programs is desired and would benefit the industry.

3.2 Analysis on the algorithms in literature

3.2.1 Analysis on Falcione & Krogh algorithm

The authors provide an example - a neutralization system with its respective Ladder Diagram control program. The process is described:

- The system starts with every actuator OFF and the tank starts empty;
- When the "start" button is pressed, valve "v1" is opened until the solution reaches "ls2" level. "v1" is then turned OFF;
- After solution reaches "ls2" level, "m" is activated. It is only deactivated after the solution goes below "ls1" level;

- Whenever the temperature of the solution is below a pre-defined value (indicated by "ts" being OFF), the heater "h" is turned ON;
- Whenever the pH of the solution is unbalanced (indicated by "as" being OFF), "v2" is opened;
- "v2" makes the solution level rise. If the solution reaches "ls3" level, "v2" is closed and "v4" is opened. Valve "v4" reduces the level of solution in the tank. When solution goes below "ls2" level, turn "v4" OFF and turn "v2" ON;
- If the temperature and pH are optimal, "v3" is opened and the level of solution is reduced. When it goes below "ls1" the tank is empty - return to the starting point and wait for the next "start" activation;
- Whenever "ts" is ON, a light "tl" becomes ON;
- Whenever "as" is ON, a light "al" becomes ON.

The end result of the algorithm is given in figure 3.1

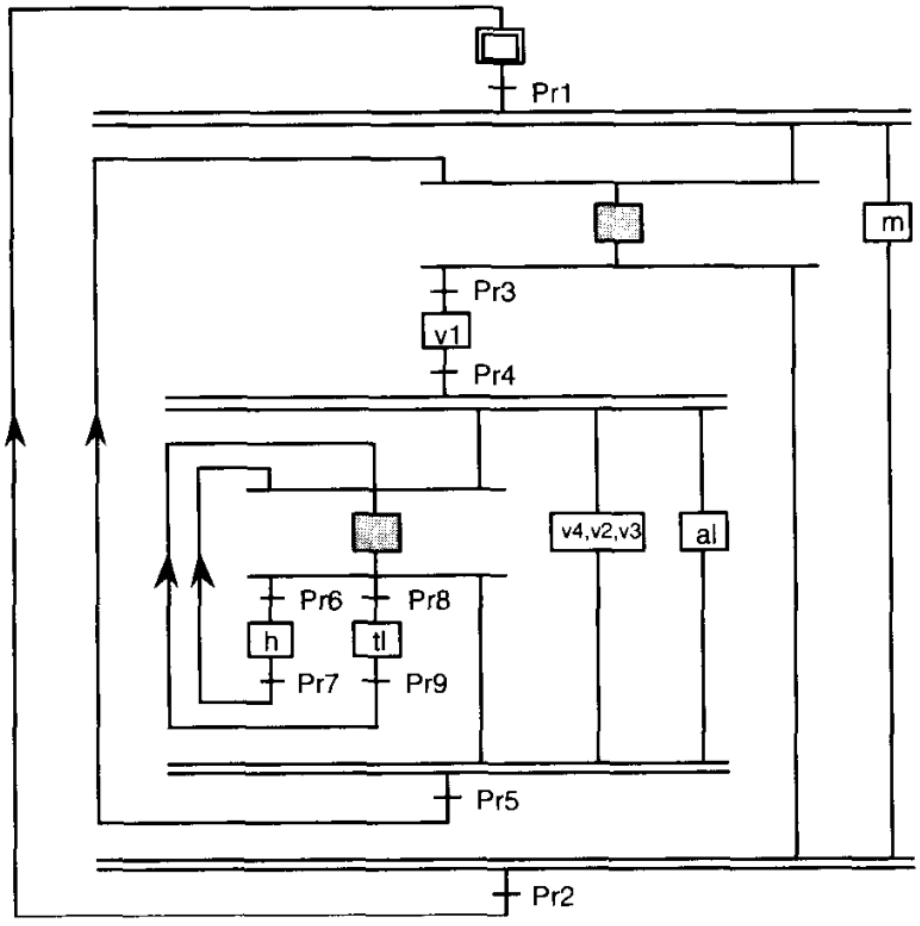


Figure 3.1: The end result of the algorithm, taken from [3]

There are two problems with the present algorithm.

First, the algorithm does not truly extract the hidden sequential logic in the Ladder Diagram. Instead, it focus on breaking the execution flow of the LD based on the relationship between rungs, forming groups of rungs that are put in sequence or in parallel. From the final SFC we cannot understand how "v2", "v3" and "v4" behave, or how "ts" and "as" being active makes "v3" turn ON. In fact, the only behaviour that matches the description of the plant is the parallelism of "m". In the end, the algorithm gives only little insight on the sequential nature of the provided example.

Second, the fact that rungs that depend on previous rungs are lumped together in a single node in the Condensed Simultaneity Graph could lead to scenarios where not even one decomposition could be made, leaving the user with a SFC with two steps, one initial, the other with all the rungs assigned to it. Consider a LD program where the pressing of a start button sets a flag which is depended upon by all the following rungs. All the rungs would then be lumped in a single node, and no decompositions could be made. Considering that, when sequential processes are described with Ladder Diagram, the dependency on previous rungs is typical, this algorithm fails to extract the sequential logic on typical cases.

Also, the algorithm does not make use of user knowledge about the plant.

3.2.2 Analysis on Zanma et al. algorithm

The main concept behind this algorithm [2] is the production of system state "snapshots" for every discrete time, guided by the plant model, which conveys information about the plant and what happens after every system state. The usage of plant model in a closed-loop with the LD sets the right scene for a proper extraction of the sequential logic, but the algorithm has multiple problems.

First, although the authors tried to formalize many concepts around their algorithm, there is a lack of definitions: Zanma et al. do not clearly define how to create SFC steps and transitions based on the $A(j)$, $B(j)$ and $B^A(j)$ sets. The algorithm is also not clearly defined in the parallel divergence extraction: at some point in the algorithm's third step, steps 1 and 2 must be applied to each sequence in the parallel divergence without considering "other sensors". The authors do not define what "other sensors" mean.

Second, divergent paths where sequence would fork are not even refered. This means that there is no specification on how to deal with forking logic situations, as the algorithm isn't capable of producing them in the SFC.

This lack of definitions makes it realy hard to implement the algorithm.

3.2.3 Analysis on the state-space approach by Babu & Nandhan

The Nandhan & Babu algorithm is a simple, easy to understand approach to sequential logic extraction. The production of a state-space considers the multiple possible sequences of states that the system can have by treating the Ladder Diagram and the plant as an asynchronous system,

where inputs can change at anytime. Due to its nature, this algorithm is of exponential complexity. For Ladder Diagrams with large number of inputs/outputs, the number of states can rise up excessively, making it too expensive to compute. The fact that all the states are extracted first, and those impossible only removed after makes it even worse, since the removal could be made at state generation time, which would save computing time.

As to the way the algorithm extracts the sequential logic, there are some problems.

Consider that the sequential logic of a Ladder Diagram being converted implies that an output X is enabled if 3 sensors A, B and C are ON and the next step enables another output Y which only needs one sensor to be enabled. Following the proposed algorithm, output Y would be enabled before output X, since Y has the least ammount of necessary input conditions, failing the extraction.

Also, the provided example does not contemplate the designing of a SFC, or a state machine. The sequential logic is described in a graph that does not resemble a state machine or a SFC and cannot be directly transformed into one.

Another problem is the conscious disregard of the information about transition between the states in the state-space. This information is useful in defining the sequence between states the system can be in.

Finally, there is an error in the heuristic rules used in the example provided by the authors that lead to a wrong reduced set of states. The author considers two sensors of being mutually exclusive, while in fact they can be OFF both at the same time. By considering that they are mutually exclusive, states where they are both OFF are incorrectly removed as "unfeasible states".

3.2.4 Conclusions taken from the analysis

- To extract the implicit sequential logic from a LD program, knowledge about the plant/process is indispensable;
- All possible sequences can be obtained if we consider the system asynchronous and toggle every input in every possible state. By doing this, we produce all the states in which the system can be;
- We can reduce the state-space by imposing certain rules while extracting the states - these rules are defined by the user and convey the knowledge about the plant;
- The information about state transitions (which state transitions to which) is necessary to a proper sequence extraction from all the possible states;
- From the set of possible state we can extract the sequence of control we are interested in if we can provide a model of the plant, in the Zanma et al. [2] did - by defining what happens if certains outputs are ON, i.e., if variable X is ON, sensor Y eventually becomes ON/OFF, etc. This information leads to the removal of unused transitions in the control sequence.

3.3 Proposed solution

The solution here proposed consists on the ideas taken from the analysis on the literature. This solution is based on the state-space approach of Babu & Nandhan [19], with changes to the way the states are extracted and sequence is constructed. The extraction of states is similar, but with the following changes:

- Feasibility of the state is defined at state creation time, when the state is being extracted. The state is then added to the state-space, but only if it is feasible;
- Information about transition between states is kept;
- The heuristic rules have more rule types than those defined by Babu & Nandhan [19] - in their paper, the authors only consider rules about input compatibility, i.e., if input X has certain value, input Y must have certain value; this solution proposal considers also compatibility between outputs and transition validation rules.

The resulting state-space will have the sequential behaviour that the LD can have with the rules imposed; imposing more rules limits even further the sequences the LD can traverse.

The solution is to be implemented as a software tool for industrial informatics, a tool that can take a IEC 61131-3 project with a LD program stored in PLCopen XML format and produce some sort of SFC (textual or graphical) that represents the LD program in the project. The tool is expected to receive as input a LD program with sequential behaviour that can be represented in SFC form. It is also expected that the LD program has its sequential behaviour hidden, i.e., that no structure to organize sequential logic is used.

3.3.1 The improved heuristic rules method

The heuristic rules are created by the user for each LD program. They represent the user knowledge about the plant. Rules can be of four types:

- Input compatibilty - a relationship between two inputs. The relationship has the form "if input I1 is true/false, then input I2 must be true/false";
- Output compatibility - a relationship between two outputs. The relationship has the form "if output O1 is true/fase, then output O2 must be true/false";
- Transition validation - a relationship between a change in the system and a certain output in the current state. The relationship has the form "if transition T has variable I with positive/negative flank, then the state from where it stems from must have output O with value true/false";
- Special button behaviour - the identification of a variable as a pushbutton. When a variable identified as a pushbutton becomes ON, the algorithm considers that the next possible transition from the state in which that variable is ON has the pushbutton becoming OFF.

When the algorithm is producing the state-space, these rules are checked against each state at the moment of its creation. If the state does not break any rule, the state is added to the state-space.

3.3.2 Evaluation of the LD program

To produce the state-space from the LD program, the latter must be evaluated as it was being executed in a PLC environment: from top to bottom, from left to right, defining the path for "power flow" from the left rail to the rest of the rung. The state is only properly defined when all of the rungs are evaluated.

To emulate the execution of the LD program internally in the software tool, an tree-based model to represent the LD is proposed.

This tree-model has a root that represents a right power rail, be it present in the LD program or not. The root has children that represent the coils. Each child node represents a single coil. The relationship between the root and its children is the connection in the LD program between the right power rail and the coils.

The nodes that represent the coils have children nodes that represent contacts. The relationship between coil nodes and contact nodes is the direct connection between them.

A pattern can be identified: the relationship between parent and child nodes is defined by the direct link between the graphical objects in a rung.

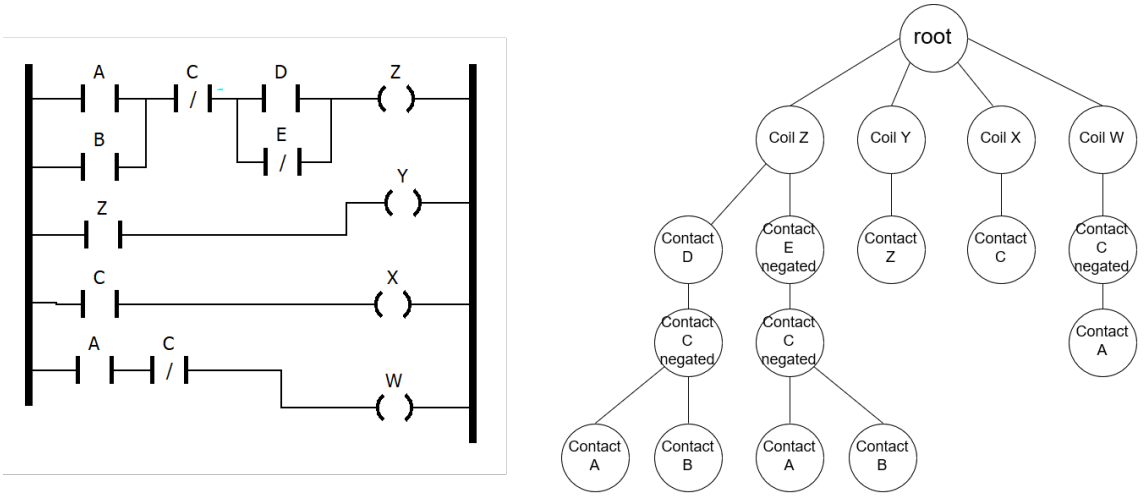


Figure 3.2: The tree model for the LD with repetition of objects

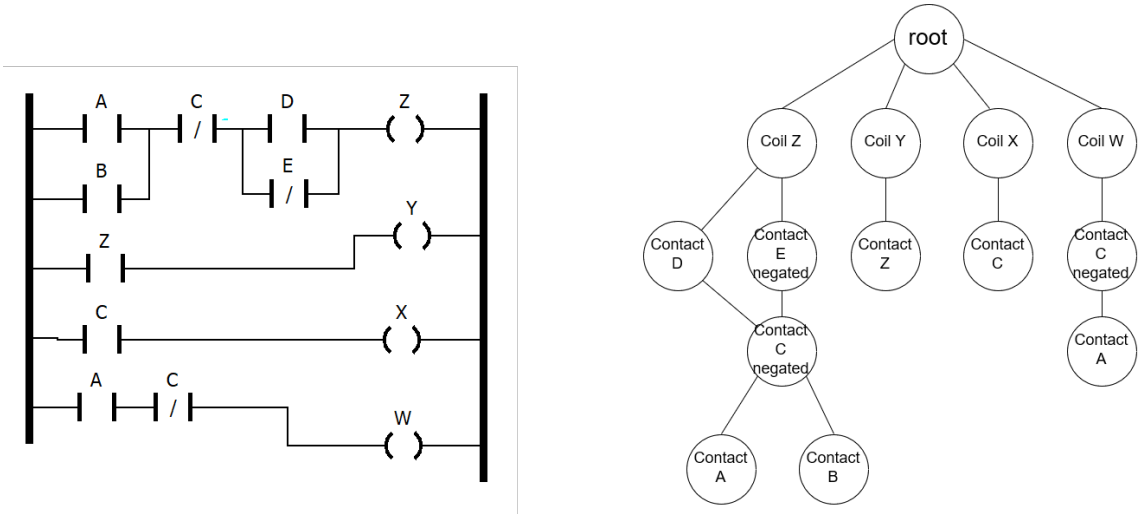


Figure 3.3: The tree model for the LD without repetition of objects

Both tree models define paths from the right power rail to the left power rail. Repetition of objects is irrelevant to the definition of paths. It only changes how many nodes and paths need to be traversed to define the value of the coil. With repetition of objects, the tree becomes the OR of the paths that connect to the left rail.

With the proposed model, executing the LD is just analysing the paths below each coil, and based on the values of the nodes in each path, define the value of the coils.

Given that the input of the software tool is a IEC61131-3 project in PLCopen XML format, this LD model must be produced from the PLCopen XML file.

3.3.3 The algorithm of extraction

With a model of the LD we can emulate the execution of the LD, and therefore, extract all possible states. With the rules defined by the user, we can simplify the state-space by removing unfeasible states at creation time. These two elements are the necessary inputs to produce a complete state-space.

The production of states uses the same strategy implemented by Babu & Nandhan [19]: by creating the first state with every input logic-0, and then, for each state in the state-space, produce more states by toggling each input, one by one, in the way illustrated in figure 2.25.

The algorithm for extraction is given in figure 3.4.

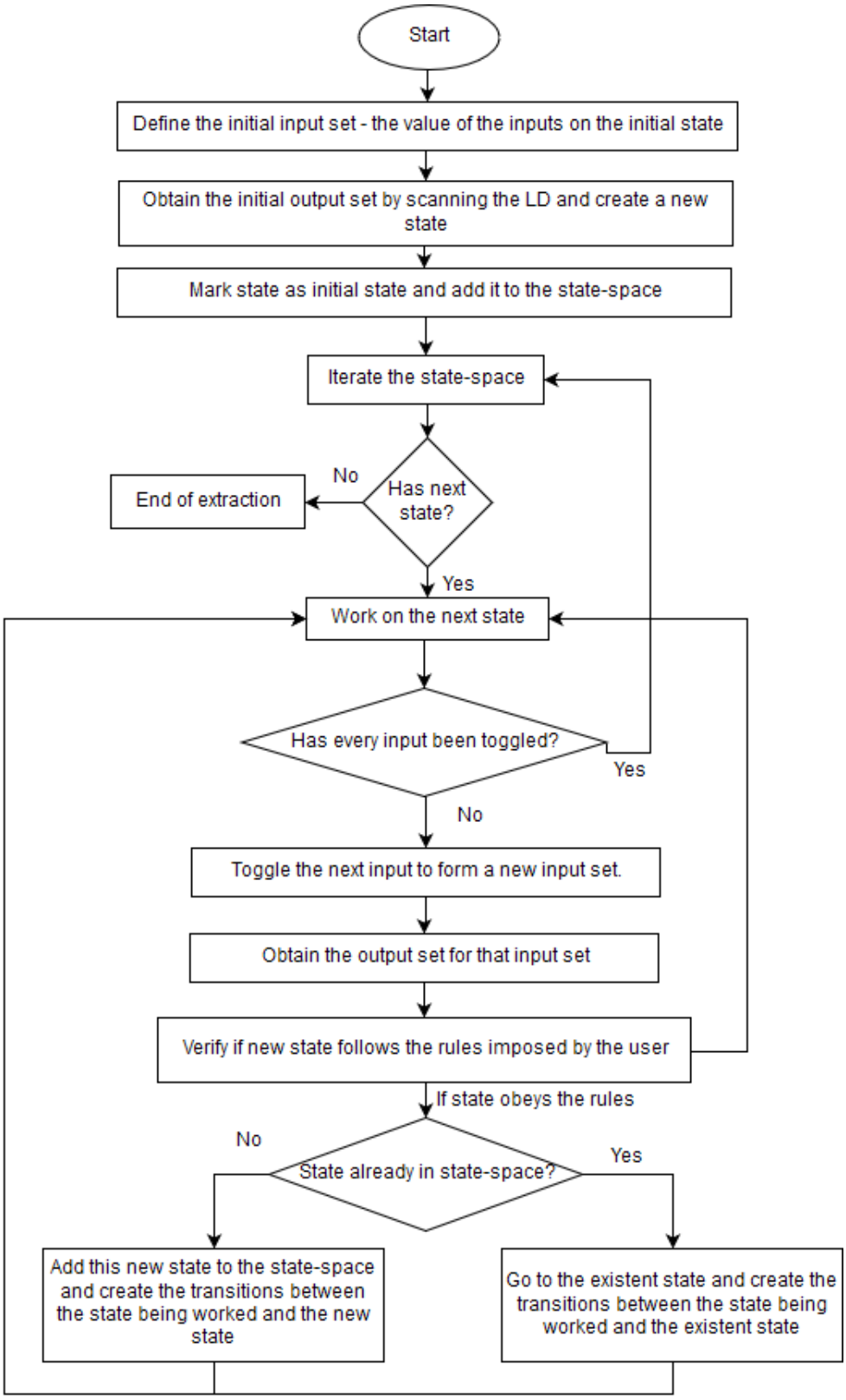


Figure 3.4: The algorithm for state-space generation

With the state-space and information about transitions, we can build the state-machine for the

respective state-space. The state-machine represents the state-space in a graphic way, exposing, in a user-friendly way, the possible states the system can be in and what events trigger the change of state.

3.3.4 Minimization of state-space

The resulting state-machine is the complete state-machine for the heuristic rules defined. This state-machine is complete in the sense that it has all the states the system can be in. But in the state-machine there are states in sequence that share the same output combination. The transitions between these states do not lead to a change in the actions on the plant, although they change the state of the plant.

When we define the sequential control tasks of a plant, we are interested in the events that lead to a change in the actions of a plant.

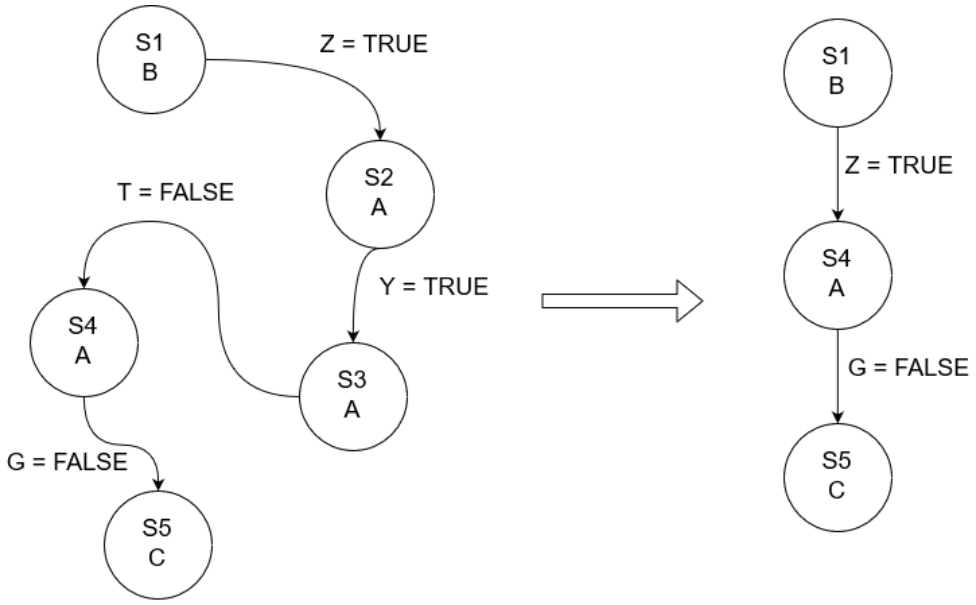


Figure 3.5: A reduction of the state-machine by compressing states with the same output combination in sequence.

These redundant states can be compressed into one state with that output combination. This way, we manipulate the complete state-machine to reduce it to a minimal state-machine where transitions lead to a change in action on the plant.

3.3.5 Extraction of parallel paths from the state-space

The sequential behaviour hidden in the LD program might describe parallel paths, where output combinations can be active at the same time. The state-machine cannot describe this behaviour since it can only have one state active at a time. If the LD describes this type of behaviour, it will be somewhat obfuscated in the state-machine, with output combinations parallel to each other being active in the same states throught the state-machine.

Pattern recognition methods can be helpful tools to extract the information about parallelism. In the work here described, a pattern is studied and a method to extract the information is explored.

Consider the example in figure 3.6.

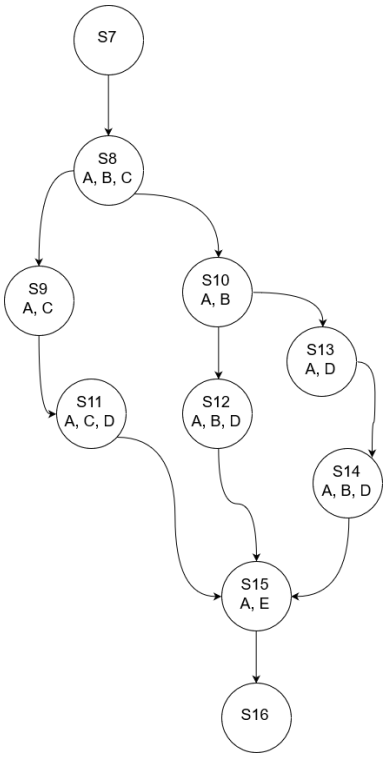


Figure 3.6: The pattern under study.

If we analyse the behaviour of the ouput A, we can see that it is common to a group of states. The activation of A and other output combinations can be traced to a single transition. And although the other output combinations change in the group, A stays the same until a single transition deactivates it and reduces the group to a single sequence of states.

Both the group and A can be isolated by those single transitions.

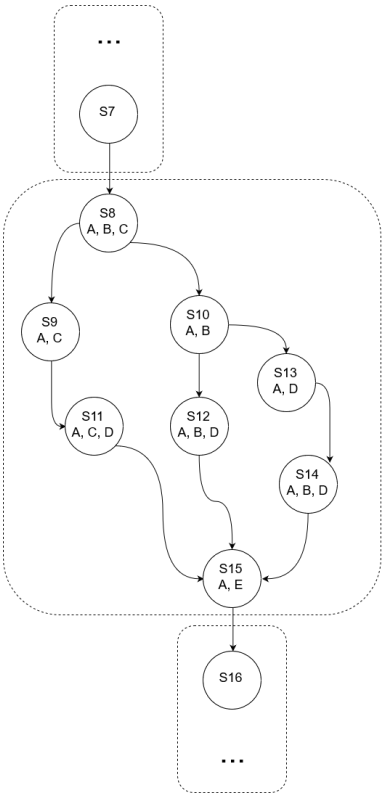


Figure 3.7: The group can be isolated from the rest of the sequence.

"A" is in parallel with the other output combinations that happen inside the group since the transitions that activate and deactivate A are the same that activate and deactivate that group of output combinations.

This pattern can be generalized to: any output combination common to a group of states where other output combinations change, and that has a single transition of activation and a single transition of deactivation, is in parallel with that group of states.

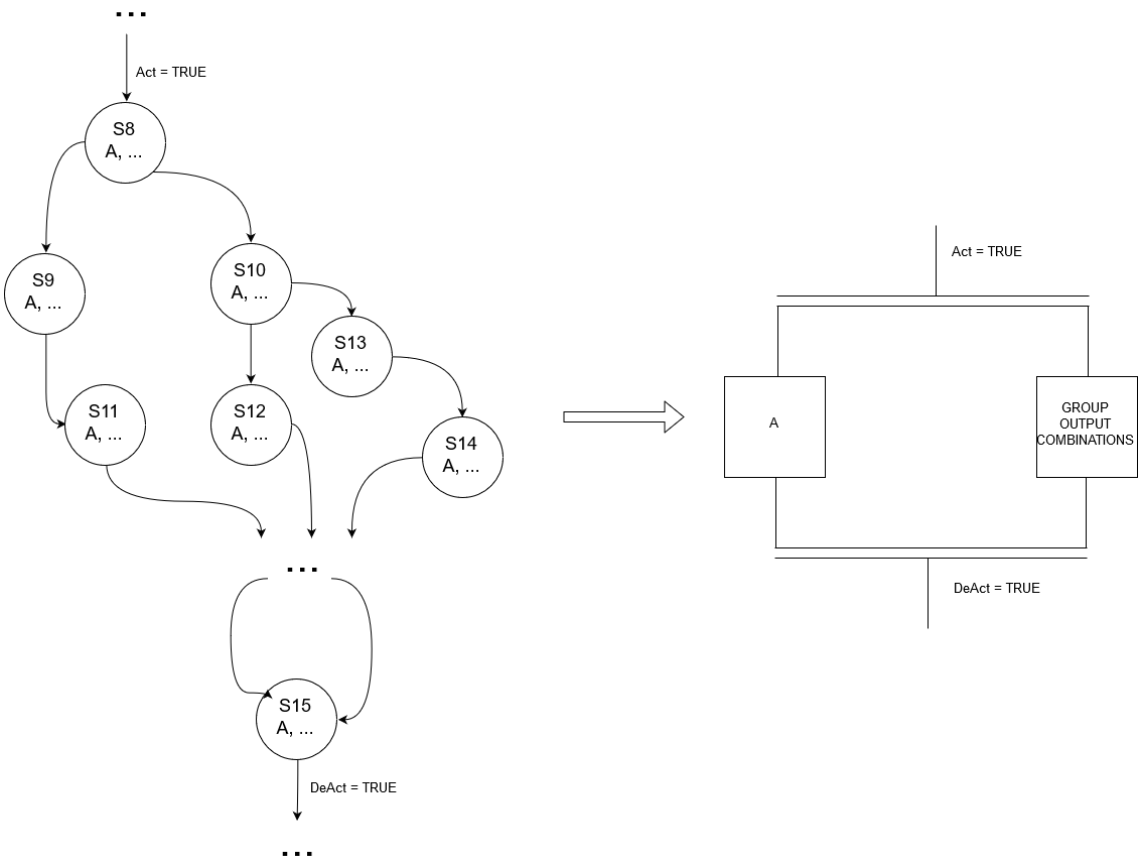


Figure 3.8: The general form of the pattern.

3.4 Work plan

Development of a software tool for the solution proposal here defined shall follow a V-Model approach - requirements analysis, software architecture (high and low level), implementation and testing with validation against the requirements.

A list of the expected tasks is defined in table 3.1. The expected dates to begin and end of each task are provided. The Gantt diagram in figure 3.9 shall be a guide throughout the development phase.

Begin date	End date	Description	Duration (days)	Task	Phase
06-02-2017	12-02-2017	Further study and analysis on the literature	7	A1	A
06-02-2017	19-02-2017	Creation of examples for the algorithms in literature	14	A2	
13-02-2017	19-02-2017	Study of XML parsing software	7	A3	
13-02-2017	19-02-2017	Study on PLCopen XML parsing	7	A4	
06-02-2017	19-02-2017	Development of Work-in-progress Article	14	A5	
20-02-2017	26-02-2017	Development of software architecture and description with UML models	7	B1	B
20-02-2017	26-02-2017	Acquisition/Creation of example programs to be used in the testing phase	7	B2	
27-02-2017	12-03-2017	Implementation of the interface, PLCopen XML parsing and LD models software	14	C1	C
13-03-2017	26-03-2017	Implementation of the algorithms for extraction	14	C2	
27-03-2017	09-04-2017	Test phase - validation and gathering of results	14	D1	D
10-04-2017	01-05-2017	Development of documentation	22	D2	

Table 3.1: Expected set of tasks during the development of the proposed solution.

The deadline for the project is estimated for the end of June. The calendarization effort in table 3.1 has an expected conclusion date to the beginning of May. The planning was made taking into account the uncertainty of the development and testing phase (C and D phases) since these are the most time-consumming tasks and only high-level planning is made. The 2-month slack accounts for possible and probable delays.

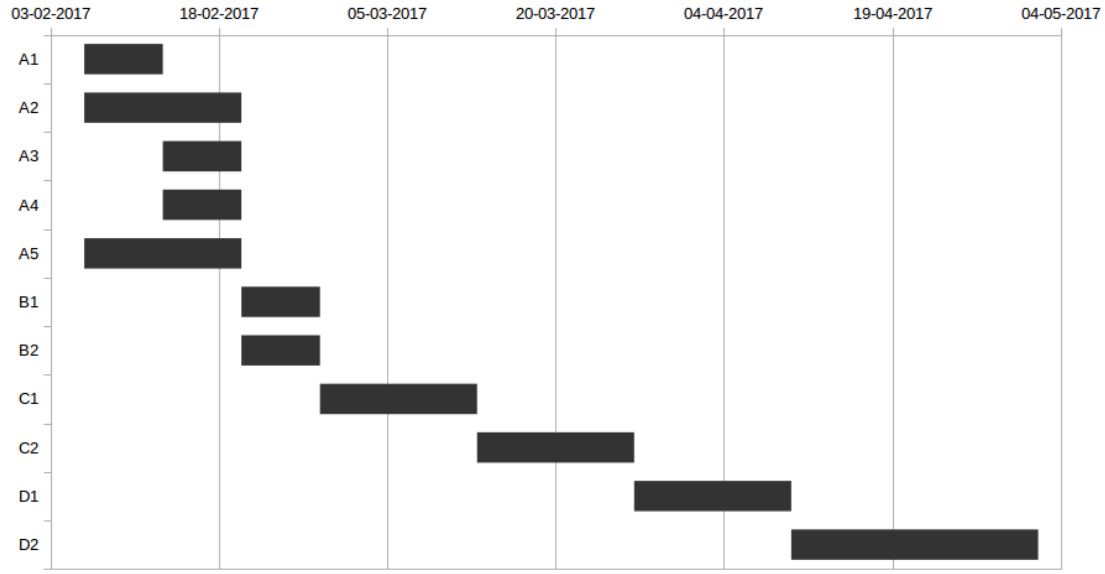


Figure 3.9: Gantt diagram for the planning in table 3.1

3.5 Development of an article on the improved state-space algorithm

As was planned, an article about the work being developed in the project was written and submitted to IEEE IES INDIN17 conference by the project author (Vitor Lopes) and the project supervisor (Mário Sousa). The article presents the algorithm without the parallel pattern identification method, since it was being developed when the article was written and submitted to the conference.

The article is entitled "Algorithm and Tool for LD to SFC Conversion With State-space Method", by Vítor Lopes and Mário de Sousa, and was submitted under IEEE INDIN17 conference.

Chapter 4

Software tool development

In this chapter, a detailed overview of the software architecture development is given. The requirements are studied and established. The high-level and low-level architectures are presented.

4.1 Requirement analysis

A requirement analysis was made to determine the functional requirements of the software tool. The list of requirements obtained considers an ideal scenario.

- The software must receive as input a IEC 61131-3 project in PLCopen XML format that can have multiple POUs;
- The user must be able to choose the POU with the LD program to be converted;
- The software must receive as input, alongside the project file, a file with the rules, preferably in XML format;
- The software must produce, as output, a PLCopen XML project file with a single POU with the SFC, preferably in graphical form, so it can be open for editing in a compliant IDE.

4.2 Software architecture

From the set of functional requirements, a high-level architecture was developed. This architecture follows a modular approach to the problem, where software modules are independent from each other. This allows for each module to be developed and tested separately, easing the development effort by focusing in each module at a time.

The architecture was developed in an object-oriented way. To implement the software, Java Programming Language was chosen.

4.2.1 High-level architecture

The high-level architecture is composed of multiple independent modules that have well defined inputs/outputs. These modules form a chain. The information is passed along that chain until a SFC is produced as output of the program.

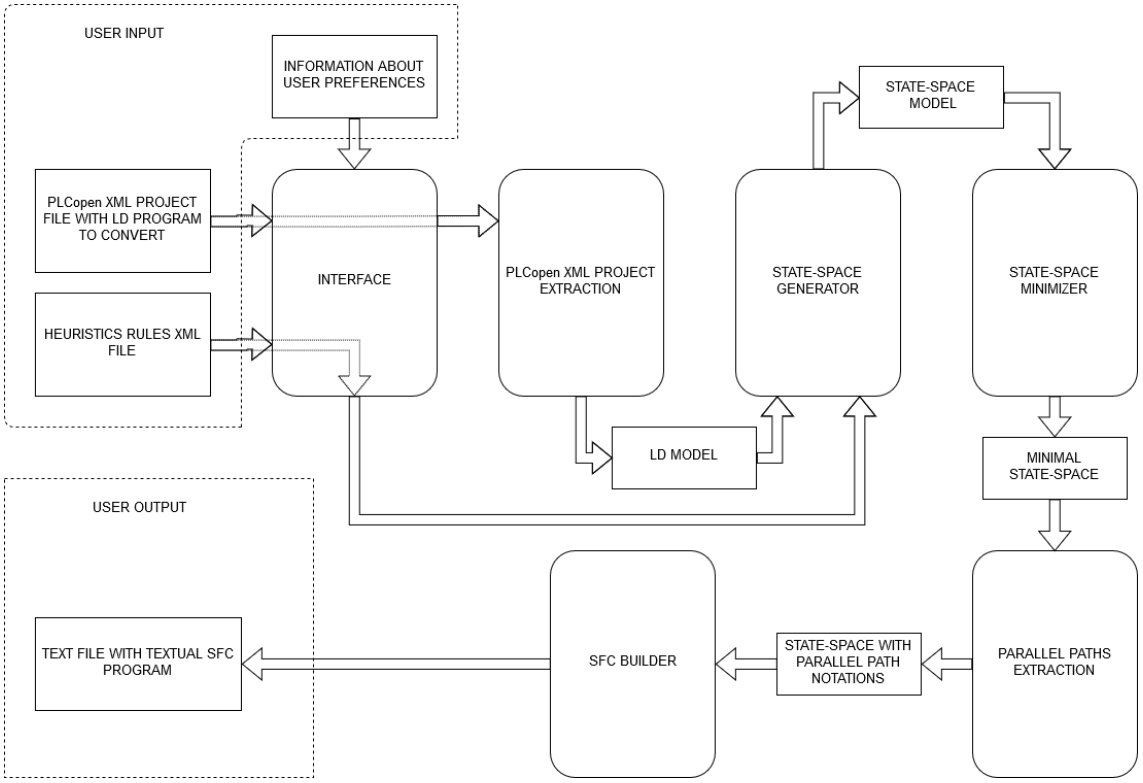


Figure 4.1: High-level architecture as a chain of independent modules.

The user input is composed of the following:

- PLCopenXML project file;
- Heuristic rules file in XML format;
- User preferences information, like the name of the POU to be extracted and filename for the output file.

This information is fed to the interface. This module provides validation on the user options. If any input is non-valid, the user is notified and the program aborts; else, the information is stored and execution control is passed to the extractor module.

The extractor module is the one responsible for creating a LD model object from the PLCopen XML project file. This module provides also validation on the project file - verification if the file is a valid PLCopen XML and if a LD program is available. In case of validation error, the user is notified and execution is aborted. In case of success, the LD model object is available when the extractor finishes execution.

At this point, execution control is passed to the state-space generator module, along with the model object and rules.

The state-space generator uses the LD model object to emulate the execution of the LD program necessary for state production, as seen in 3.3.3. The state-space generator runs the algorithm described in chapter 3 and stores the information in a State-space Model object. If any error occurs during the state-space creation, execution is aborted and user is notified.

The resulting state-space model object is available at the end of the state-space generator execution. This model stores information about every possible state and transition according to the rules provided.

The following two modules in the chain (State-space minimizer and Parallel path extraction modules) take the state-space model object and manipulate it according to the pattern identification methods described both in 3.3.4 and 3.3.5. If any error occurs during the execution of any of these modules, execution is aborted and the user is notified.

The last module - SFC builder - is the module in charge of transforming the manipulated state-space (minimal state-space with parallel paths annotations resultant of the work of the previous two modules) into a plain-text file with SFC in textual form.

All the modules, with the exception of the interface, are objects with fields and methods that allow each module to properly function. The interface is implemented at the level of the "main" method, the point of execution start. The "main" method runs the logic necessary to handle interface, errors and execution control passing to the methods of each module according to the high-level chain defined here.

The following sections provide a low-level view of each model and module.

4.2.2 XML file structure for the heuristic rules

The heuristic rules XML file has a simple structure to represent the various types of rules the user can create. These rule types are defined in 3.3.1.

The root element is the "rulebook" element. This element does not possess any attributes. The root can contain four types of child elements, without any of them being obligatory. The child elements match the four rule types:

- <ii> tag to define an input compatibility rule;
- <oo> tag to define an output compatibility rule;
- <io> tag to define transition validation rules;
- <pushbutton> to define special button behaviour.

Each of these child elements have no attributes. Each one of the "ii", "oo" and "io" elements have two child elements of type "variable". The "pushbutton" element has only one child element of type "variable". In a pushbutton definition, the "value" attribute in the "variable" element is left in blank.

A "variable" element is an empty element with two attributes: "name" and "value". The "name" attribute represents the variable name, and the "value" attribute represents the value of the variable in the rule.

For example, if we have the following rules, we can represent them as described in listing 4.1.

- "if input A is true, then input B must be false"
- "if output C is false, then output D must be true"
- "if transition T has variable E with positive flank, then the state from where it stems must have output F with value true"
- "variable G is a pushbutton"

Listing 4.1: The representation of the example rule in XML format

```
<rulebook>
  <ii>
        <variable name="A" value="true"/>
        <variable name="B" value="false"/>
  </ii>

  <oo>
        <variable name="C" value="false"/>
        <variable name="D" value="true"/>
  </oo>

  <io>
        <variable name="E" value="true"/>
        <variable name="F" value="true"/>
  </io>

  <pushbutton>
        <variable name="G" value=""/>
  </pushbutton>
</rulebook>
```

The order of the elements if important to the proper definition of the rule: the element that defines name and value for the "A" variable in the rule must come first than the element that defines the name and value for the "B" variable.

A schema definition file was not developed.

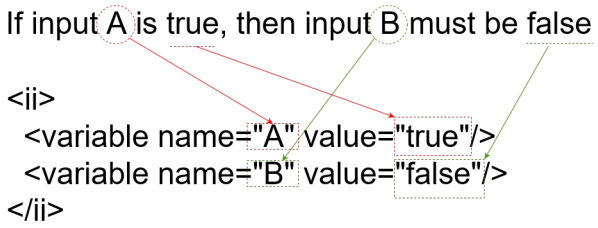


Figure 4.2: Rule structure in XML notation.

4.2.3 LD Model

The LD Model is the group of Java classes described in the UML diagram in appendix A. The main class is the LadderDiagram class. This class stores 4 fields: "inputs", "outputs" and "vars" are of List<String> type and store the names of the variables in the Ladder Diagram program. The "root" is of LadderComponent type. This "root" LadderComponent object is the root of the tree as defined in 3.3.2.

The LadderComponent class represents a node in the tree. LadderComponent objects store information about the children, the parent and the information about the graphical object ("data" field) of that node (coil or contact).

The information about each graphical object is stored in an object of type "LadderObject", as "Coil" or "Contact" objects (these objects inherite from the "LadderObject" class).

In 3.3.2 two types of trees are considered: with and without repetition of objects. The software model here defined allows both options.

4.2.4 State-space Model

The State-space Model, like the LD Model, is a group of Java classes under the same package that provide the fields and methods to create and manipulate a state-space as defined previously in chapter 3. The UML class diagram for the State-space can be found in appendix A.

The main class is the "StateSpace" class. This class stores a collection of states and the variables that are present in the LD program. Methods are provided to deal with the state-space.

The "State" class represents a state in the state-space. It mainly stores information about inputs and outputs status, state name (unique identifier in the state-space) and the set of incoming and outgoing transitions. The other flags are for use with the manipulation of state-space algorithms.

The "Transition" class stores information about the variable that changes and its flank, and information about the state from where the transition starts and the state where it ends.

The "Pair" class is a helper class to define a variable name with a boolean value associated with it.

The "HeuristicRule" class is independent from the previous classes. It represents a rule as defined in 3.3.1. Its type, variables and values are extracted from the XML file with the rules defined by the user. It is, therefore, a small model of a rule.

4.2.5 The interface module

The interface with the user is made through the command line. Passing information (project file, rules file, user preferences) is made via argument passing to the program. The set of arguments defined is composed by the following arguments:

- "-help" - the -help argument enlists all the arguments available to the user;
- "-filename projectFile.xml" - used to point the location of the project file that has the LD to be converted. In this case, "projectFile.xml" in the same directory as the executable would be the file to be worked on;
- "-o outputFile.txt" - used to point the location of the output file to be written; this file shall contain the textual SFC program;
- "-pou name" - in the case of the project file having multiple POUs, the user must provide the name of the POU to be converted - this can be done by using the "-pou" argument with the name of the pou after;
- "-rules rulesFile.xml" - used to point the location of the rules file;
- "-ld filename.txt" - used to export a text file with DOT[20] instructions - the output file describes the LD tree model of the LD submitted for conversion in a graphical form;
- "-ss filename.txt" - used to export a text file with DOT[20] instructions - the output file describes the state-space in a graphical form.

When the program starts execution, arguments are checked. The software verifies if arguments are correctly used. If any error occurs in the verification phase (bad arguments, etc) execution is aborted and the user is notified. Two arguments are absolutely necessary and the program won't progress without them: project file (argument "-filename") and rules file (argument "-rules"). In case of no compliance, the user is informed of the error and asked to try again.

As it stands, the software tool is in a Java "jar" container, so usage is as follows:

```
java -jar converter.jar [args]
```

After sucessful argument checks, the program continues. Execution control is handed to the extractor module.

4.2.6 The extractor module

The extractor module is the one responsible for extracting the information from the project file and creating a LD tree model. The model allows both forms (with and without repetition of objects - see 3.3.2). The algorithm here described builds a tree model with repetition of nodes.

It takes, as input, the name of the project file and rules file. It outputs a "LadderDiagram" object that represents the LD tree model and a collection of "HeuristicRule" objects that represent the rules file. See 4.2.3, 4.2.4 and appendix A for "LadderDiagram" and "HeuristicRule" classes.

To extract the information from the XML file, JDOM is used. JDOM is a Java-based solution for handling XML files and provides methods to deal with XML in DOM fashion, by creating a tree for the XML document. DOM is explored in 2.3.5.

Step 1: the algorithm starts by verifying if the files (project file and rules file) exist at the location provided by the user. If the software fails to find the files, the program terminates in error and the user is informed; else, the algorithm goes to the second step.

Step 2: the algorithm proceeds with the building of XML Document structures, one for each input file. If no error occurs while building the in-memory structures to deal with the file, the PLCopen XSD (XML Schema Definition) file is loaded. The XSD is used to validate the project file, as it is expected that the user provides a valid PLCopen XML file. If the file fails validation, it is not a proper PLCopen XML file - execution is aborted and user is notified of the error. If it is a valid file, the algorithm advances to the third step.

Step 3: this step consists in extracting the rules from its XML file. First, the root element is extracted. Then all its children is obtained and stored in memory. Iteration over the children collection allows accessing each element. These child elements are the "ii", "oo", "io" and "push-button" elements, the ones that contain information about the rules. For each child element in that collection, the rule type (name of the element), and the attributes of its children (the "variable" elements) are extracted and a "HeuristicRule" object is created and added to a collection. At the end of the iteration, a collection with multiple "HeuristicRule" objects that represent the rules XML file is available.

Step 4: the next step is to find the POU to convert. The PLCopen XML file is explored. If no "pou" elements are found inside "pous" element (under project/types/) error is signaled and execution is aborted; else all "pou" elements that have a "LD" element inside their "body" element are put in a temporary collection. If, when all "pou" have been iterated, there is only one "pou" with LD elements, that POU is the one to be converted. If more than one "pou" with LD elements exist, the name of the POU provided by the user in the command line is used to identify the right POU. If no name was provided, error is returned and execution aborted. At the end of the step, if a POU is identified with success, that POU is kept in memory and used in the next step.

Step 5.1: first, all the coils and contact elements under the "LD" element in the body of the chosen POU (pou/body/LD/) are extracted and kept in collections - one for coils, other for contacts. Then, the right power rail element is extracted. The right power rail element has important information for the ordering of the coils (from top to bottom) since all the coils connect to the rail. These connections are elements in themselves and have position information and identify the coil being connected to the rail. By ordering the connections by vertical position value, we can also order the coils. Then, for each coil a node in the tree model is created under the root, as child node. For each node, a "Coil" object, with all the necessary information about the coil (type of coil and variable assigned to it), is created.

Step 5.2: For each coil added to the tree root as child, the sub-tree of contacts is built. This is done with a simple recursive algorithm:

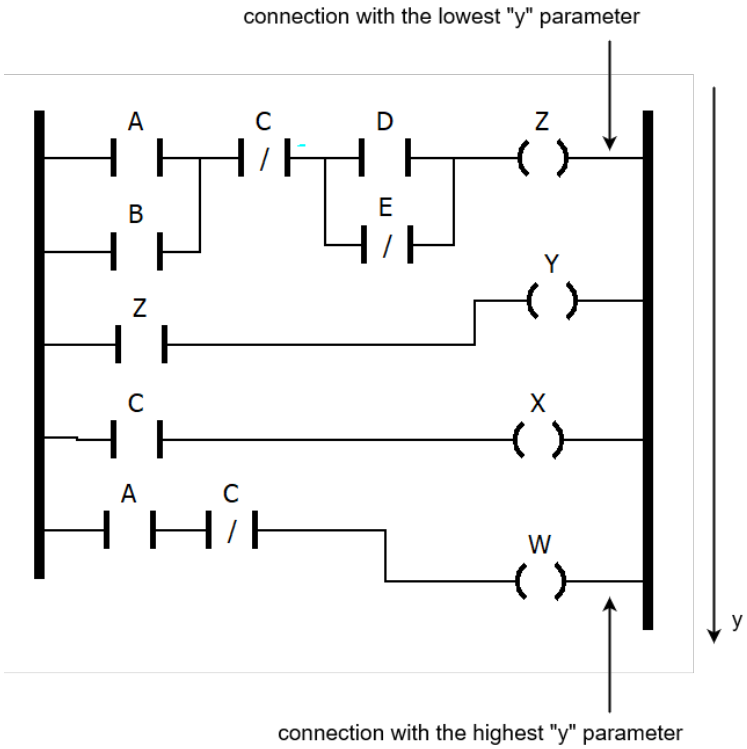


Figure 4.3: The order of the coils can be obtained from the vertical position of the connection to the rail

5.2.1: starting with the coil, its connection elements that define the links to the left of the graphical coil object are retrieved. The connections identify the graphical objects connected imediately to the left of the coil. For each connection, that is, for each graphical object connected to the coil from the left links, a child node is created under the coil node in the tree model. A "Contact" object is assigned to the "data" field of each node with the information about the contact (type and variable assigned to it).

5.2.2: the process is repeated for the each graphical object at the left of the previous object in a recursive way: while calling the method to extract the graphical objects at the left of the coil, recursively call the same method to extract the graphical objects at the left of the graphical objects that are on the left of the coil, and so on until there are no more contacts at the left.

When each node is created, the variables are also extracted and kept in the "inputs", "outputs" and "vars" fields of the "LadderDiagram" object.

The algorithm only supports the extraction of contacts and coils. LD programs with function blocks will not be properly extracted. At the end of the algorithm, if there is no problem with the LD, no errors occur and the tree model is complete and available. In case of error, execution is aborted and the user is notified.

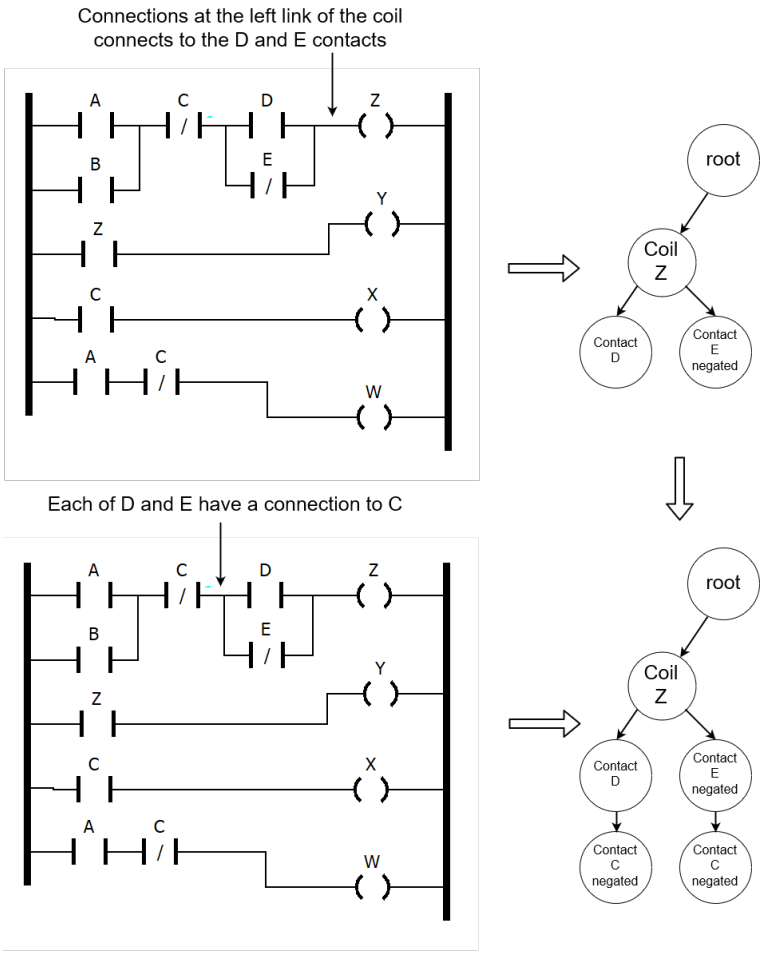


Figure 4.4: Each iteration of the algorithm gets the next objects at the left

4.2.7 The state-space generator module

The state-space generator module is the one responsible for building the state-space from the LD tree model and heuristic rules set extracted in the previous module. It implements the algorithm described in 3.3.3.

The algorithm starts by creating the first state by creating an input set (List<Pair> type, see 4.2.4) with every input variable (obtained from the LD model) with logic-0. The LD model is evaluated to determine the output set. With both sets, the initial state is formed. This state is marked as initial by setting the "initial" flag. Then, starting with the initial state, other states are produced by toggling inputs one by one, as explained in 2.4.3.1. To produce the next state, the new input set (with one input toggled) and output set of the previous state is used while evaluating the LD to produce a new output set. If a state with the same input and output set already exists in the state-space, no new state is added. Instead, it is created a transition from the previous state to the existent state. When no new states can be formed the algorithm ends.

Evaluation of the LD model The evaluation method takes two sets: an input set and an output set. These sets have boolean values assigned to each variable. The evaluation uses the LD model - the "LadderDiagram" object produced in the previous module - to determine a new output set.

To determine the boolean value for each coil, the contact sub-tree is recursively analysed. A coil can only be assigned boolean "true" if a path of contact objects are closed to allow "power flow" to reach the coil. To that end, the children of the coil are analysed to see if "power flow" can reach the coil through them. But to find if "power flow" can reach those contacts, the contacts on the left (the child nodes below) must be checked. The same process repeats for contacts that have contacts at their left links. When a node has no children, it means that that coil is at the leftmost of the rung and is, therefore, at the source of the "power flow". The process is repeated for every coil in the LD tree model.

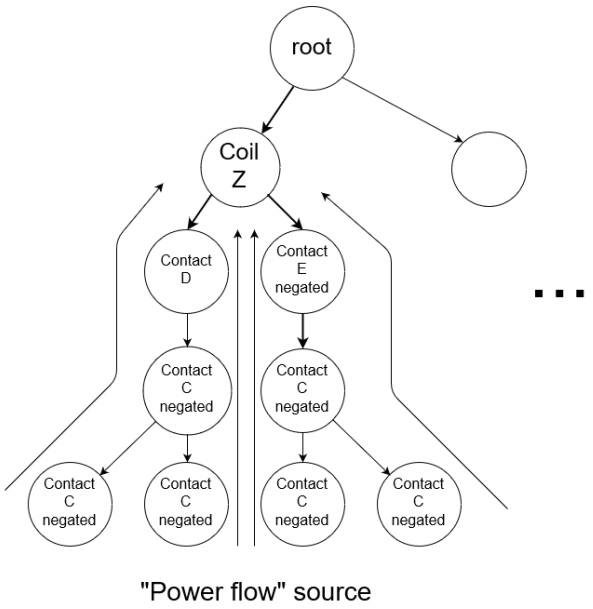


Figure 4.5: Example with possible four paths for "power flow" to reach the coil

Determining the feasibility of a state So that the state can be added to the state-space, it must comply with all the rules set by the user. These rules are represented in a collection of "HeuristicRule" objects created in the previous module.

The method to check for feasibility takes a state and:

- first checks all the input compatibility rules - if any forbidden combination of inputs is found in that state, the state is deemed unfeasible; else, the method advances to the next check;
- then checks all the output compatibility rules - if any forbidden combination of outputs is found in that state, the state is deemed unfeasible; else, the method advances to the next check;

- then, the transition rules are check - if the variable and value of the transition that leads to the state being checked and the outputs of the previous state form a forbidden combination, the state is deemed unfeasible.

The verification of pushbuttons is made in the following manner: if the state being checked has a variable identified as "pushbutton" with boolean value "true", then, if the previous state has that same variable "true", the state is deemed unfeasible because variables marked as "pushbutton" must become "false" after being "true".

Output of the module If the process was successful, at the end of the algorithm a "StateSpace" object is available; else, error is returned and the execution is stopped. This "StateSpace" object represents the state-space (see 4.2.4).

4.2.8 The minimizer module

The minimizer module takes the "StateSpace" object produced in the previous module and manipulates it according to the method described in 3.3.4. It outputs the same "StateSpace" object after completing all the minimization procedures on it.

The algorithm goes state by state and for each state, it checks the next states. If a next state has the same output combination as the state, the transitions are changed in the following manner:

- Transitions that start at the nextState are changed so that they now start at the state;
- Transitions that arrive at the nextState are changed so that they now arrive at the state.

The next state is then marked for removal. Transitions that end up starting and arriving at the same state are also marked for removal. The figure 4.6 illustrates the process.

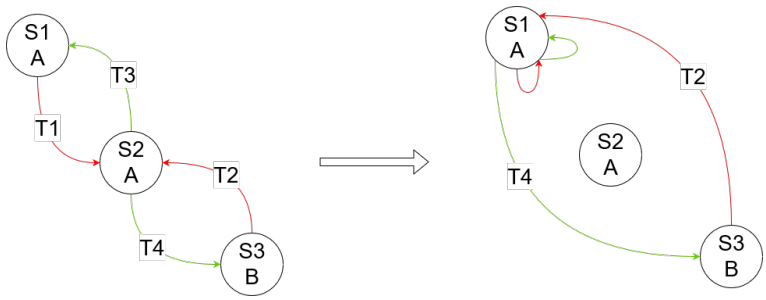


Figure 4.6: Illustration of the minimization procedure

After all the states have been searched, the states and transitions marked for removal are removed. This results in a reduced state-space.

4.2.9 The parallel extraction module

The next module is the parallel extraction module. This module is responsible for identifying the pattern described in 3.3.5. It takes as input the reduced state-space from the previous module and outputs a state-space with the changes according to the pattern extraction method in 3.3.5.

The algorithm to find the pattern is as follows:

Step 1: Get the output variable set from the "StateSpace" object. For each variable do steps 1.1 through 1.5. If there are no more output variables to process, end the algorithm.

Step 1.1: Create two "pools" of states as illustrated in figure 4.7 ; one with all the states where the output is ON and the other with all the states where the output is OFF.

Step 1.2: Check how many transitions exist from one pool to another, in both directions. If there is only one transition in each direction, go to step 1.3; else, return to step 1 and process the next variable.

Step 1.3: Check how many states there are in the pool where the output is ON. If there is more than one state in that pool, advance to step 1.4; else, go to step 1 and process the next variable.

Step 1.4: At this point, we have found a variable that shows the same behaviour as described in 3.3.5. Now, we create a new state with that output ON. A copy of the transition that goes from the pool where the output is OFF to the pool where the output is ON is made and the state where the transition arrives is changed to the new state, i.e., the copy is changed so that is points to the new state. The same is done for the transition that goes from the "OFF pool" to the "ON pool", with the exception that the copy is changed so that the transition starts from the new state. The copies are then linked to the original transitions and vice-versa. This means that they form two parallel paths and should be treated as parallel divergence and synchronization. Go to step 1.5.

Step 1.5: turn the output OFF in the states where it is ON, with the exception of the new state. Go back to step 1 and process the next variable.

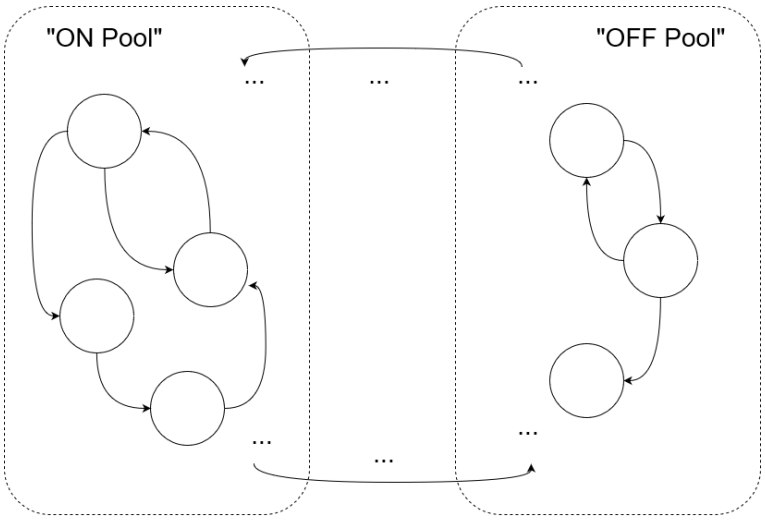


Figure 4.7: Illustration of the parallel path extraction procedure

4.2.10 The SFC builder module

The last module takes the "StateSpace" object after it being manipulated in the previous modules and produces a textual representation of the SFC program in a plain-text file. If the user did not provide a name as argument to the interface ("-o name"), an automatic name is generated with the formatted date and time.

The SFC is a direct representation of the final state-space (the reduced one, with parallel annotations). The initial state is written to the file as an "INITIAL_STEP". The other states are written as normal "STEP".

The actions (the status of the outputs) are written in ST at the textual definition of the step. It behaves like normal N actions.

When writting the steps to the file, its respective transitions are also written.

Chapter 5

Results

In this chapter two examples are explored with a complete follow-through of the algorithms steps. A discussion of the resultant software tool is made with the help of the examples.

5.1 Example: Falcione & Krogh neutralization tank

In 3.2.1, an analysis on the algorithm of Falcione & Krogh is made with the help of the results the authors obtained in their paper. The neutralization system is explained in the following reiterated points:

- The system starts with every actuator OFF and the tank starts empty;
- When the "start" button is pressed, valve "v1" is opened until the solution reaches "ls2" level. "v1" is then turned OFF;
- After solution reaches "ls2" level, "m" is activated. It is only deactivated after the solution goes below "ls1" level;
- Whenever the temperature of the solution is below a pre-defined value (indicated by "ts" being OFF), the heater "h" is turned ON;
- Whenever the pH of the solution is unbalanced (indicated by "as" being OFF), "v2" is opened;
- "v2" makes the solution level rise. If the solution reaches "ls3" level, "v2" is closed and "v4" is opened. Valve "v4" reduces the level of solution in the tank. When solution goes below "ls2" level, turn "v4" OFF and turn "v2" ON;
- If the temperature and pH are optimal, "v3" is opened and the level of solution is reduced. When it goes below "ls1" the tank is empty - return to the starting point and wait for the next "start" activation;
- Whenever "ts" is ON, a light "tl" becomes ON;

- Whenever "as" is ON, a light "al" becomes ON.

The Piping & Instrumentation Diagram (P&ID) is supplied by the authors and is available in figure 5.1.

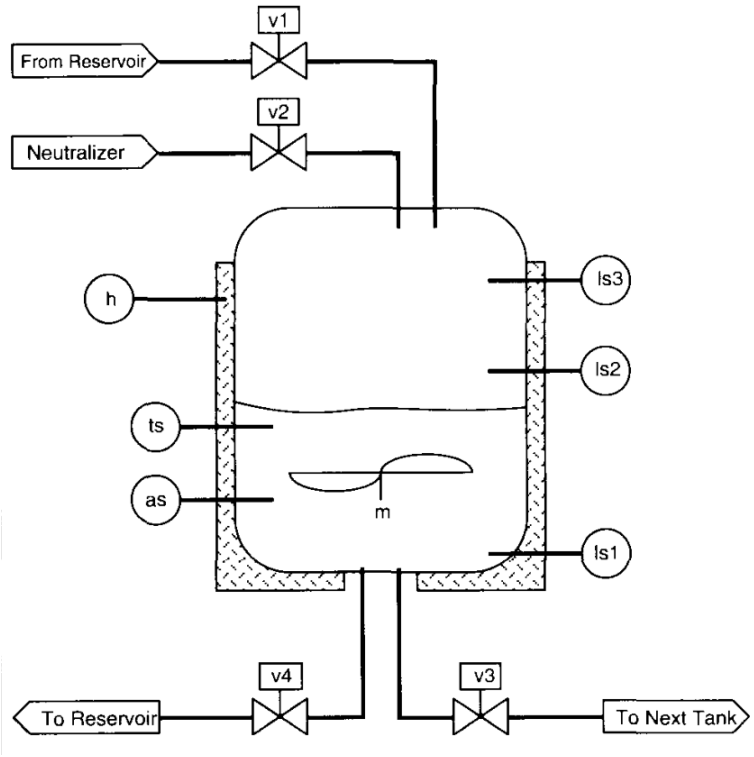


Figure 5.1: Piping & Instrumentation Diagram for the A. Falcione's and B. Krogh's example, taken from [3]

5.1.1 Obtaining a PLCopen XML project file and defining rules

The Ladder Diagram program for the neutralization system is provided by the authors and is available in figure 5.2. This LD program was replicated with the Beremiz IDE so that a PLCopen XML file could be produced with the example used.

A set of rules was developed to model the plant. The set of input compatibility rules is given by:

- If "ls2" is true, then "ls1" must be true - this rule reflects the nature of the tank: if "ls2" is true it means the water reached ls2 level, which implies that "ls1" is submerged;
- If "ls3" is true, then "ls2" must be true - like the previous rule, this rule is about the water level. The combination of these two rules imply that "ls3" cannot be true without "ls2" and "ls1" being true.

The set of transition validation rules is given by:

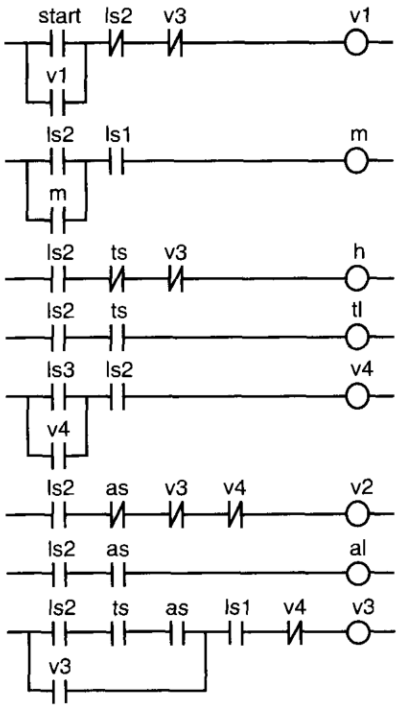


Figure 5.2: Control program for the example above, written in Ladder, taken from [3]

- If a transition has variable "ts" with positive flank, then the state from where it stems must have output "h" with value true - this rule reflects the relationship between the temperature sensor, the rise in temperature and the presence of the heater; whenever the heater is turned on, the temperature rises and "ts" can turn on;

- If a transition has variable "as" with positive flank, then the state from where it stems must have output "v2" with value true - this rule results from the fact that "as" becomes true when the pH of the solution is at a certain point and that "v2" opens a valve that drops neutralizer in the tank. When neutralizer is added, the pH solution approaches the optimal point and "as" can become true;

- If a transition has variable "ls3" with positive flank, then the state from where it stems must have output "v2" with value true - this rule represent the fact that the solution level only reaches the "ls3" level if "v2" is turned on. No other valve can make the solution rise up to that level;

- If a transition has variable "ls2" with negative flank, then the state from where it stems must have output "v3" with value true

- If a transition has variable "ls2" with positive flank, then the state from where it stems must have output "v4" with value true - "ls2" becoming false means that the solution level dropped below that level and only "v3" or "v4" can empty the tank;

- If a transition has variable "ls2" with positive flank, then the state from where it stems must have output "v1" with value true
- If a transition has variable "ls2" with positive flank, then the state from where it stems must have output "v2" with value true - the last two rules reflect the fact that only "v1" and "v2" valves can fill the tank;
- If a transition has variable "ls1" with positive flank, then the state from where it stems must have output "v1" with value true - this one reflects the fact that only "v1" can fill the tank initially, before the solution reaches "ls2" level;
- If a transition has variable "ls1" with negative flank, then the state from where it stems must have output "v3" with value true - "ls1" going false means that the tank is completely empty; only "v3" can completely empty the tank.

Finally, "start" is defined as a pushbutton. No output compatibility rules are defined.

5.1.2 Building of the LD tree model

After feeding both the project and rules files to the interface of the software tool, the set of algorithms in chain are started. The first module - the extractor - takes the project file and builds a tree model of the LD program. A plain-text file with DOT[20] language instructions that represents the tree can be obtained by using the argument "-ld name". The graph can be visualized using Graphviz software [4]. The LD tree model for the example is given in figure

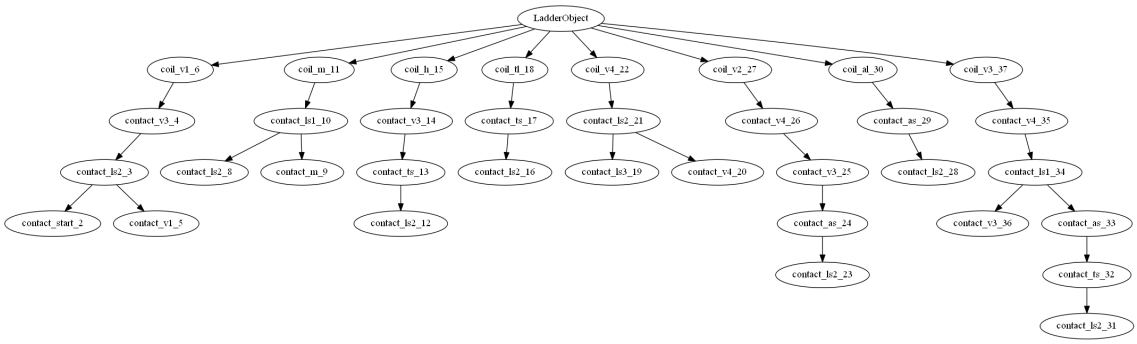


Figure 5.3: The LD tree model developed from the PLCopen XML project file; visualization produced with Graphviz [4]

5.1.3 Building the state-space with the rules defined

With the LD tree model defined and the ruleset, the state-space generator produced the complete state-space given limitations imposed in the rules. This complete state-space can be seen in figure 5.4.

In the graphic visualization of the state-space, the square marks the initial state. Starting from the initial state, we can see how the sequence progresses. When start is pressed, "v1" is turned on.

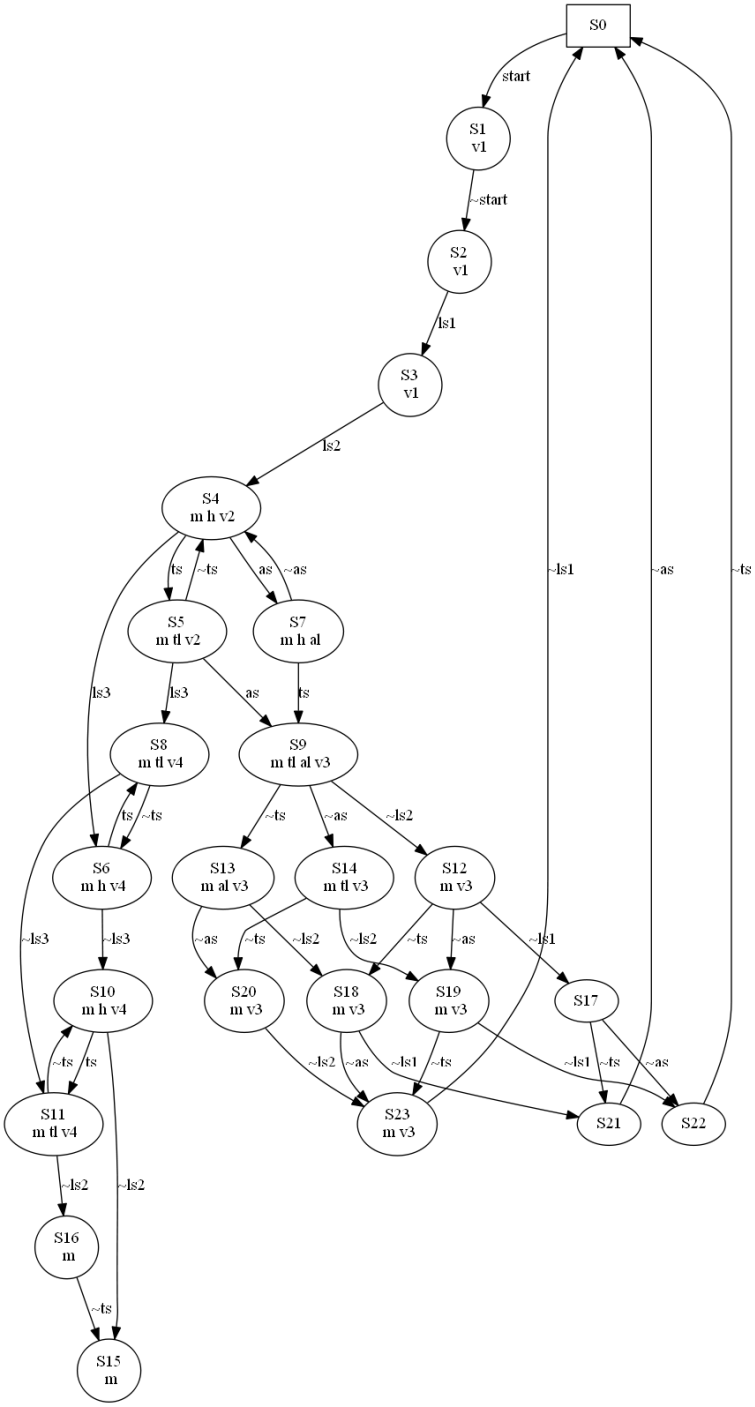


Figure 5.4: The complete state-space for the neutralization system; visualization produced with Graphviz [4]

Then, start is expected to be released, since it is a pushbutton. We can then see the evolution of the solution level: first "ls1" turns on and then "ls2", as expected. When the solution reaches that level, the heater "h", the mixer "m" and the valve "v2" are actuated. Then, many events can occur: if the temperature reaches the optimal point (signaled by "ts") the heater is turned off and the light

"tl" is turned on; if the pH is at optimal point, "v2" turns off and the light "al" is turned on; if the level reaches "ls3" before the solution, "v2" is turned off and "v4" is turned on. We can see that, when "ts" and "as" are true, the system turns "v3" on. The system returns to the initial state as the liquid level goes down and "ts" and "as" turn off. The mixer is kept working until "ls1" becomes false.

Analysing the left side of the diagram we can see that there is a deadlock, states from where the system cannot leave. The system enters states S16 and S15 in the aftermath of the solution reaching the "ls3" level. After that, "v4" is turned on, as previously seen, and we can see the evolution in the liquid level: first "ls3" goes off and then "ls2". When "ls2" turns off, the system enters a deadlock with only the mixer working.

From the definition of the neutralization system, this should not occur. If the liquid level goes below "ls2" after "v4" is opened it is expected that "v2" turns on again and the system returns to a state where it is trying to reach pH optimal point, the state S4 or S5. This failure can be traced down to the LD program used by Falcione & Krogh [3]. There is only one rung responsible for the actuation of "v2" and that rung does not consider the need for "v2" to be active after "v4" is opened and the level goes below "ls2".

This error is not easily detected with the LD alone. With the state-space, the error is exposed and the origin can easily be traced.

A correction was made to the LD program to fix this problem. The modified LD program is used in the next section.

5.2 Example: Modified neutralization tank

As seen previously, the original LD has an error that compromises the control of the system. A correction was made. The modified LD can be seen in figure 5.5.

The process was repeated with the new LD program, using the same rules. A new complete state-space was generated and can be seen in figure 5.6. The new state-space no longer has a deadlock when the liquid level goes below "ls2". "v2" is turned on and, when "ls2" becomes true again, the system returns to S4 and S5 states, resuming normal functioning.

The next producedure is the minimization of the state-space. The minimization module takes the state-space and reduces it based on the heuristic methods proposed in 3.3.4 and 4.2.8. The resultant state-space is available in figure 5.7.

The reduced state-space does not reveal all the sequences. It considers only the events that lead to a change in the actuation of the plant.

With the reduced state-space we can see better that the system could also end up in a lock, in state S22. When the liquid level goes below "ls2" after "v4" is opened, "v2" is turned on. At that moment and right before the liquid goes above "ls2", the optimal pH can be reached, leaving the system locked in S22. Proper corrections to the LD program would need to be made to fix this problem.

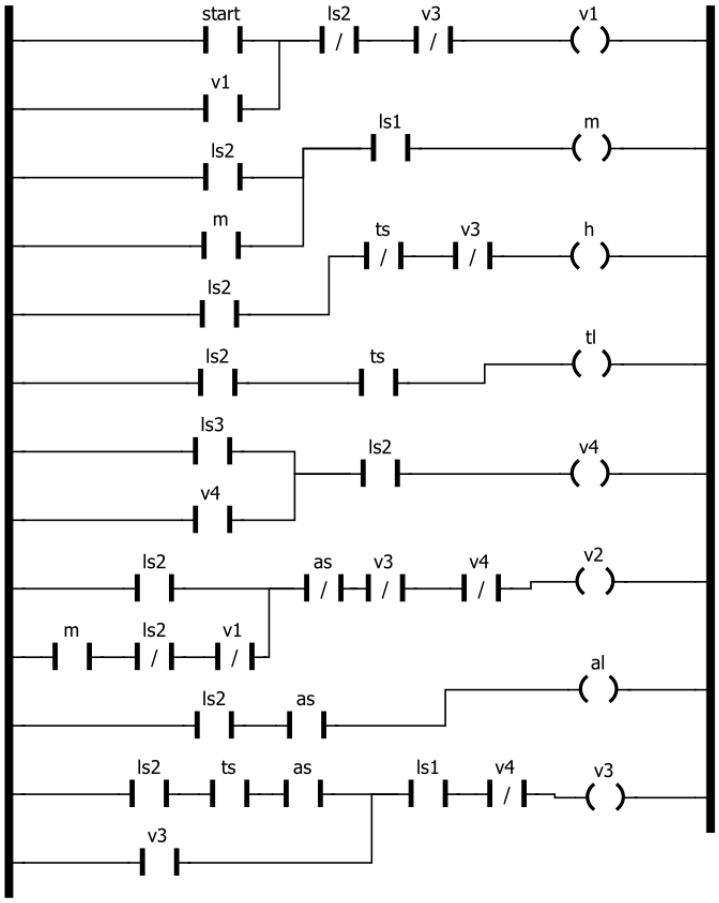


Figure 5.5: The modified LD program, produced with the Beremiz IDE editor

After applying the minimization procedure, the parallel paths extraction procedure is applied according to the methods explained in 3.3.5 and 4.2.9. The resulting state-space with a parallel path can be seen in figure 5.8. As we can see in the reduced state-space, "m" variable is active in a group that has only one entry transition and one exit transition. This pattern is the same as the one approached in chapter 3.

In the diagram, the transition copies are separated and states S4 and S4_1 can be active at the same time. Therefore, this diagram is not a proper state-machine. Internally, the copied transitions are linked together so that the last module correctly represent them as simultaneous divergence and simultaneous convergence.

Finally, the last module takes the state-space diagram and produces a plain-text file with textual SFC that is a direct representation of the state-space diagram with parallel paths. The output plain-text textual SFC can be found in appendix B.1.

5.3 Discussion

The developed software tool, designed to convert LD programs into SFC programs by extracting the hidden sequential logic, is, as it stands, capable of taking a project file in PLCopen XML format and extract the state-space for that LD based on rules defined by the user. These rules convey the knowledge about the plant needed to properly extract the sequential behaviour. But although it can produce a state-space, it cannot fully extract the parallel paths from it, recognizing only one type of parallel path pattern.

If we analyse the reduced state-space with parallel paths we see that "h" and "v2" share some sort of parallel relationship, evident in the set of states {S4, S5, S6, S7, S8}. We can see that "ts" becoming true/false and "as" becoming true/false events do not affect each other. In that set of states, a change in "ts" lead to a change in "h" and nothing more. The same can be said of "as", leading to a change in "v2". The events "as" and "ls3" can be understood as divergent, since one or the other happens, leading to different sequences. Both "as" and "ls3" only affect "v2". Therefore, from the state-space we can extract that the parallel structure between "h" and "v2" has the form illustrated in figure 5.9.

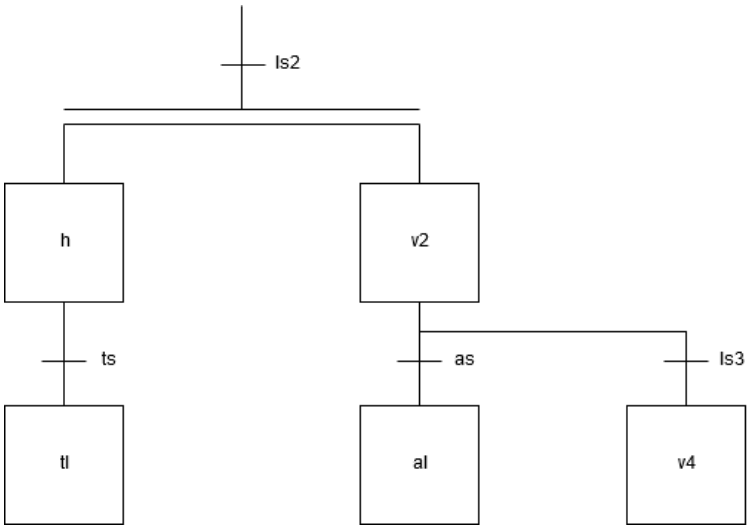


Figure 5.9: Parallel structure between "h" and "v2"

After "ts" and "as" become true, the sequence sychronizes and "v3" is turned on. Meanwhile, "m" is kept in parallel with all of this until "ls1" becomes false, where both "m" and "v3" turn off, i.e., "m" and "v3" synchronize and the next is the initial step, where the process can be started once again.

The resultant SFC program is much more complex than it could ideally be due to the inability to extract this information. Therefore, we can conclude that more pattern identification methods are needed to properly extract information about parallel paths.

Another critical part of the software tool is the possibility of explosion in the state-space size. The software tool can handle small LD programs, but as the size increses so does the number of states in a exponential way. To try and limit the growing number of states more rules are needed

to limit what states can be created. Even then, if the program is complex, the number of states will be high.

To properly understand the impact of this aspect, testing the complexity of the algorithm is needed. With proper testing we can determine the limits of the algorithm.

Relative to the rules, defining a variable as a pushbutton can be useful to restrict the number of states. In the previous example, if "start" wasn't defined as pushbutton, the resultant complete state-space would have double the size, since there would be another sequence like the one extracted but with "start" always "true". Then, since "start" could become "false" at any time, there would be a lot of transitions between the states where "start" is true and those where "start" is false.

While practical to keep the state-space as minimal as possible, there is a problem with this strategy: if the system goes through various events in quick fashion such that they happen before the button is released, the resulting state-space does not properly represent the real control sequence.

Another aspect is that the software does not take into account that the system could have a different initial state than every input logic-0. The user cannot define the initial state, which results in the real initial state not being the first step in the SFC.

As to the requirements, the tool somewhat achieves the requirement of IEC 61131-3 project in PLCopen XML as input format, since it supports PLCopen XML but only supports LD with only coils and contacts. The requirements of user being able to choose the LD and being able to receive plant knowledge through the rules are achieved. The software tool fails to meet the last requirement: producing graphical SFC in PLCopen XML format, only producing textual SFC in plain-text file.

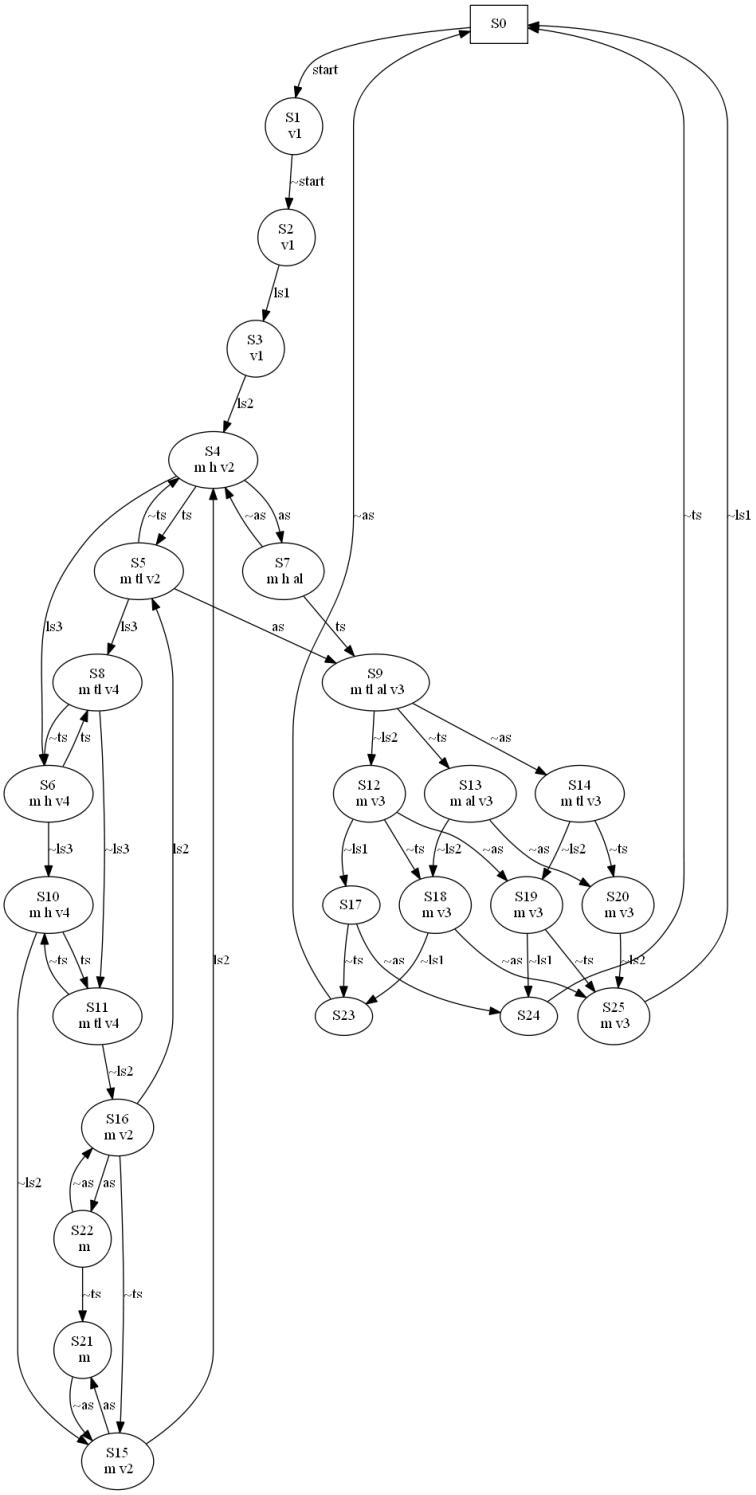


Figure 5.6: The new complete state-space; visualization made with Graphviz[4]

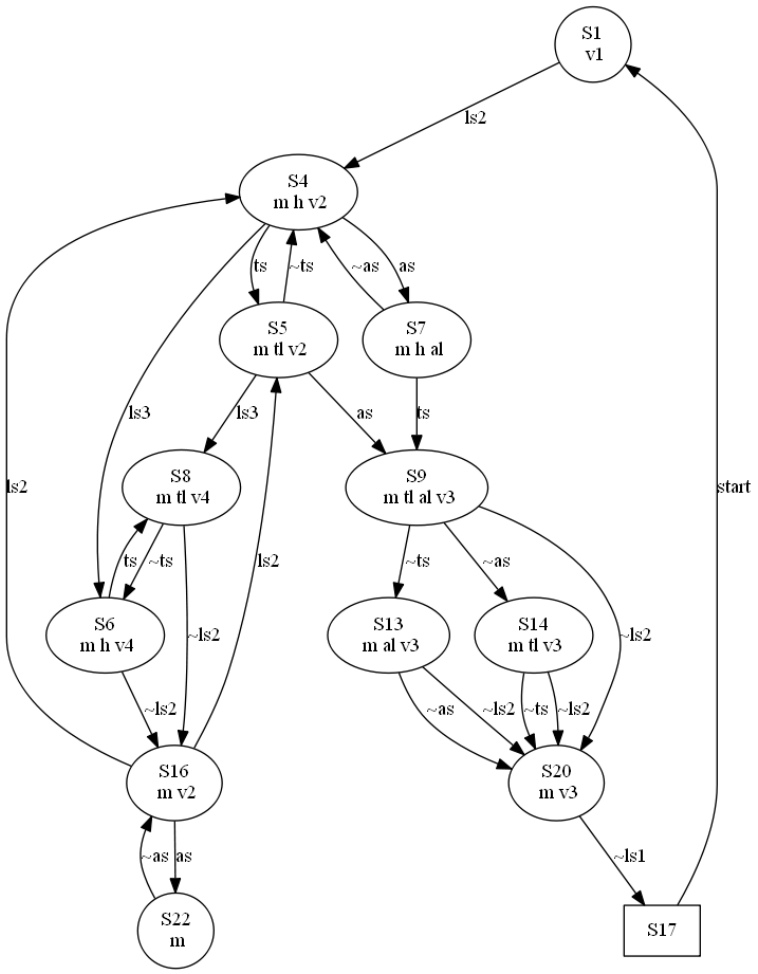


Figure 5.7: The reduced state-space; visualization made with Graphviz[4]

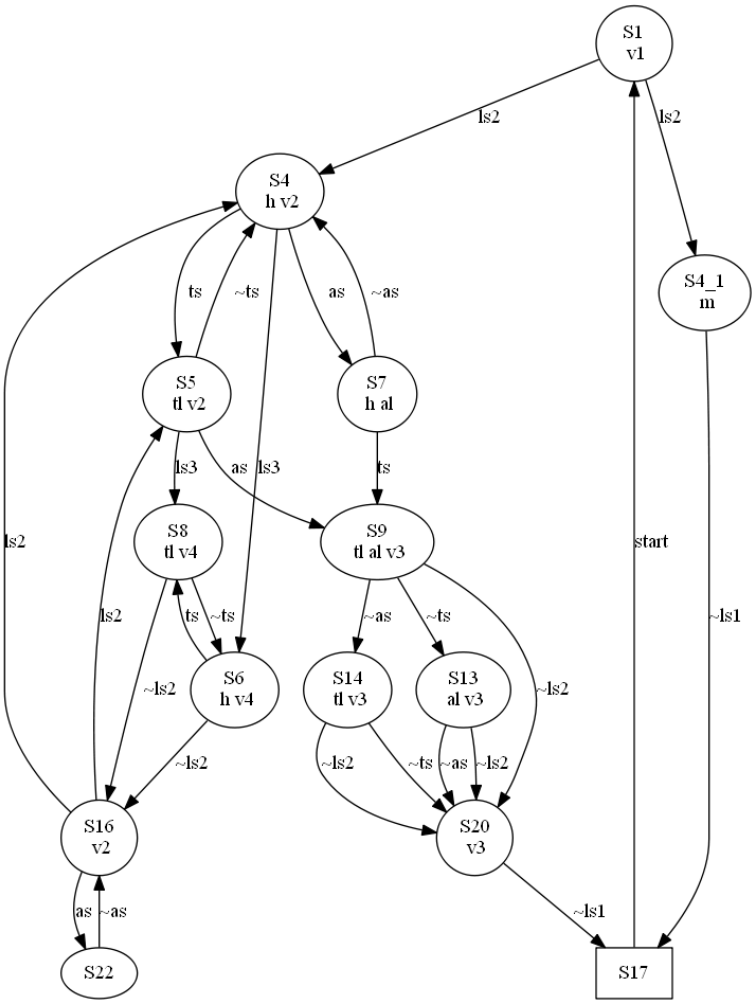


Figure 5.8: The reduced state-space with a parallel path; visualization made with Graphviz[4]

Chapter 6

Conclusion

In this section a conclusion on the work developed in the project is provided. Topics for future work are also presented.

The work here described concerns the development of a software tool for the extraction of hidden sequential logic in Ladder Diagram programs and represent that same sequential logic in a SFC program.

To that end, the IEC 61131-3 standard was studied, with focus on LD and SFC languages' structure and rules and in the software structures defined.

Along with IEC 61131-3, XML and XML parsing was studied so that understanding of PLC open XML, a standard that allows structuring of IEC 61131-3 projects in an open and interchangeable way, was deepened.

To tackle the problem in context, three different algorithms in literature were studied and analysis was made on them. The first algorithm studied presented some problems and inability to handle the majority of LD programs. The second algorithm presents a formal algorithm, but it is not well defined in some parts, leading to a high difficulty of implementation. The third algorithm presents an approach based on state-space extraction that is not well defined in respect to the extraction of the sequential logic.

Based on the analysis' conclusions, a set of algorithms is proposed. The solution is based on the state-space approach, but with changes and improvements on the state-space creation. An high-level modular architecture, with specification on how each module is implemented, is presented.

The proposed software solution was implemented in Java Programming Language. The software tool is capable of receiving an IEC 61131-3 project in PLCopen XML format and output a textual SFC program. The program can also output graphic representations of the models used and interchanged between the modules of the software.

An example of implementation is explored and some results of the implementation are discussed that raise topics for the future work on this project. Some problems about the software tool were discussed, but in general the tool meets the objectives, failing only to produce a PLCopen XML file as output with the SFC program.

6.1 Future Work

Although it meets, in general, the proposed objectives, a lot of work can be done to improve the software tool. Some of the topics to explore are:

- Add the ability to process LD programs with Functions and Function Blocks and to emulate their execution. To that end, improvements over the model, extraction module and state-space generator module are needed;
- Add the ability to identify and process LD programs structured for sequential logic with step logic. These structures carry all the needed information to build the SFC and the creation of the state-space might not be needed;
- Add the possibility for the user to define the initial state of the plant;
- Improve the parallel path extraction module, adding the ability to recognize much more patterns in the state-space and, therefore, extract much more information about the parallelism in the system;
- Improve the module responsible for building the output SFC, so that it can produce graphical SFC in a PLCopen XML format;
- Develop studies and tests on the complexity of the proposed set of algorithms.

Appendix A

UML models

A.1 Ladder Diagram model class diagram

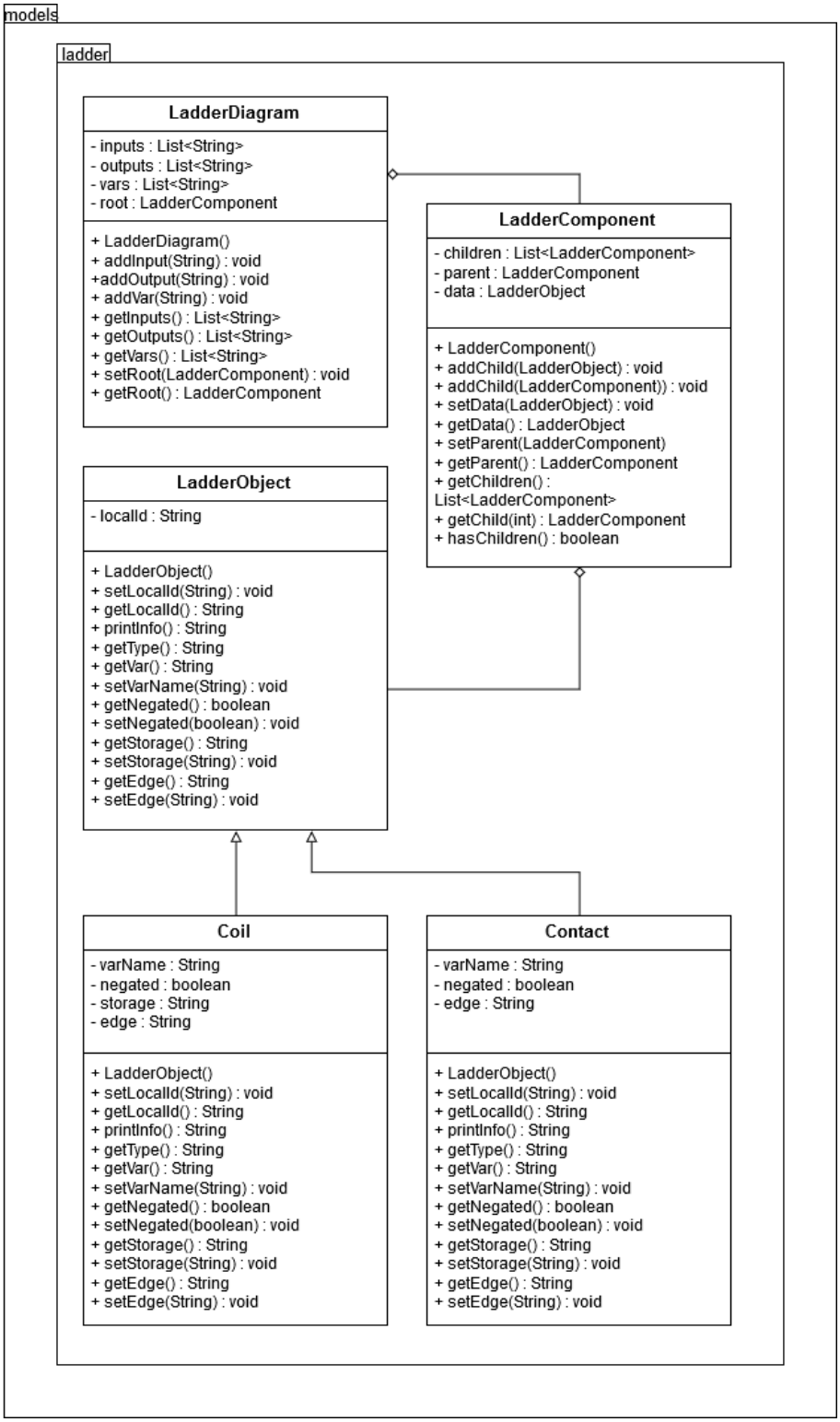


Figure A.1: Final UML class diagram for the Ladder Diagram internal model

A.2 State-space model class diagram

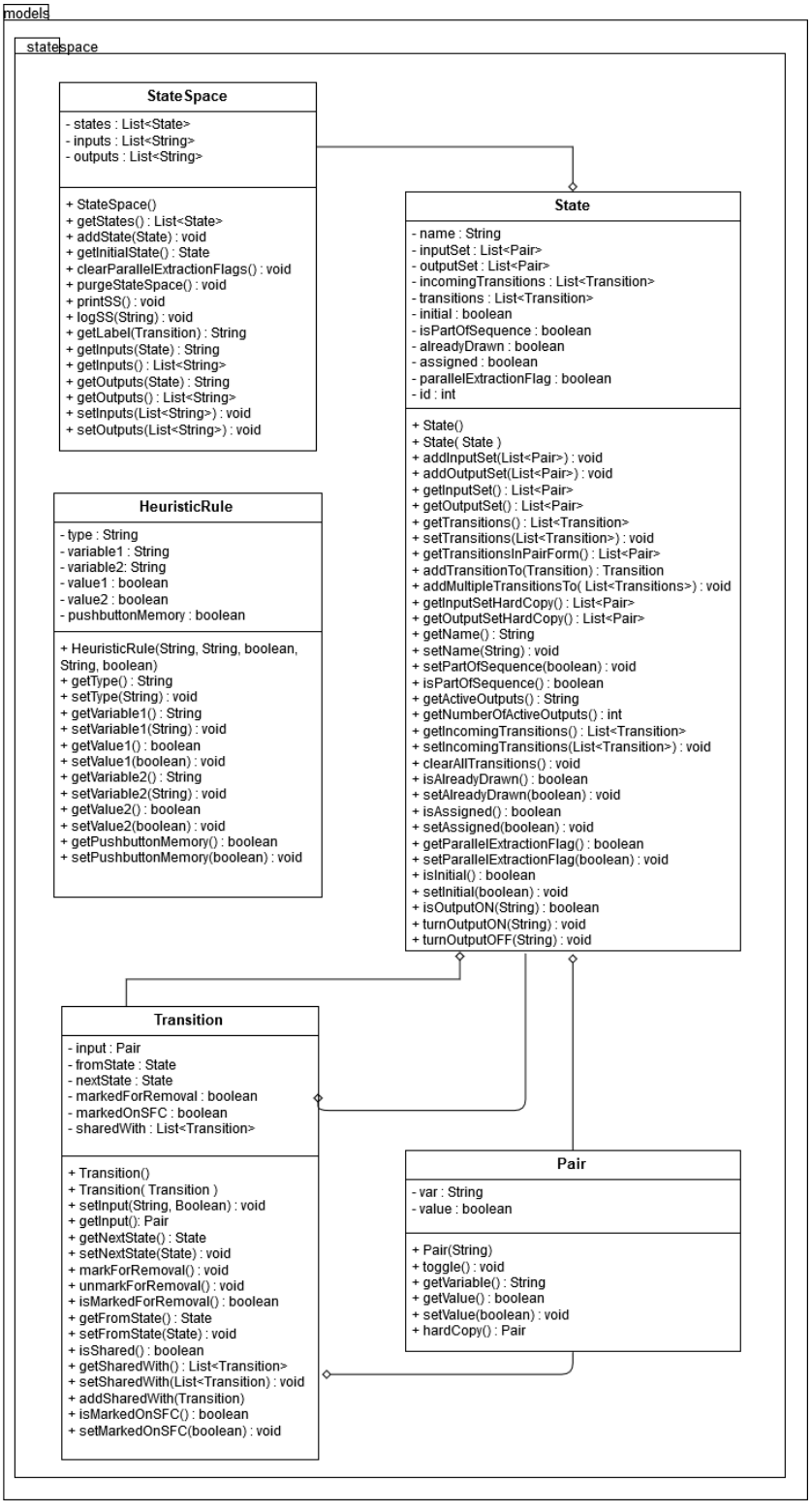


Figure A.2: Final UML class diagram for the State-space Model

Appendix B

Software tool output SFC

B.1 Textual SFC as output of the software tool

Listing B.1: Output of the software tool for the example in 5.2 - textual SFC

```
STEP S1:
v1:= TRUE

END_STEP

TRANSITION FROM S1 TO (S4,S4_1)
:= ls2
END_TRANSITION

STEP S4:
h:= TRUE
v2:= TRUE

END_STEP

TRANSITION FROM S4 TO S5
:= ts
END_TRANSITION

TRANSITION FROM S4 TO S6
:= ls3
END_TRANSITION

TRANSITION FROM S4 TO S7
```

```
:= as
END_TRANSITION

STEP S5:
t1:= TRUE
v2:= TRUE

END_STEP

TRANSITION FROM S5 TO S4
:= NOT ts
END_TRANSITION

TRANSITION FROM S5 TO S8
:= ls3
END_TRANSITION

TRANSITION FROM S5 TO S9
:= as
END_TRANSITION

STEP S6:
h:= TRUE
v4:= TRUE

END_STEP

TRANSITION FROM S6 TO S8
:= ts
END_TRANSITION

TRANSITION FROM S6 TO S16
:= NOT ls2
END_TRANSITION

STEP S7:
h:= TRUE
a1:= TRUE

END_STEP
```

```
TRANSITION FROM S7 TO S9
:= ts
END_TRANSITION

TRANSITION FROM S7 TO S4
:= NOT as
END_TRANSITION

STEP S8:
t1 := TRUE
v4 := TRUE

END_STEP

TRANSITION FROM S8 TO S6
:= NOT ts
END_TRANSITION

TRANSITION FROM S8 TO S16
:= NOT ls2
END_TRANSITION

STEP S9:
t1 := TRUE
a1 := TRUE
v3 := TRUE

END_STEP

TRANSITION FROM S9 TO S20
:= NOT ls2
END_TRANSITION

TRANSITION FROM S9 TO S13
:= NOT ts
END_TRANSITION

TRANSITION FROM S9 TO S14
:= NOT as
```

```
END_TRANSITION

STEP S13:
a1:= TRUE
v3:= TRUE

END_STEP

TRANSITION FROM S13 TO S20
:= NOT ls2
END_TRANSITION

TRANSITION FROM S13 TO S20
:= NOT as
END_TRANSITION

STEP S14:
t1:= TRUE
v3:= TRUE

END_STEP

TRANSITION FROM S14 TO S20
:= NOT ls2
END_TRANSITION

TRANSITION FROM S14 TO S20
:= NOT ts
END_TRANSITION

STEP S16:
v2:= TRUE

END_STEP

TRANSITION FROM S16 TO S5
:= ls2
END_TRANSITION

TRANSITION FROM S16 TO S22
```

```
:= as
END_TRANSITION

TRANSITION FROM S16 TO S4
:= ls2
END_TRANSITION

INITIAL_STEP S17:

END_STEP

TRANSITION FROM S17 TO S1
:= start
END_TRANSITION

STEP S20:
v3:= TRUE

END_STEP

TRANSITION FROM (S20,S4_1) TO S17
:= NOT ls1
END_TRANSITION

STEP S22:

END_STEP

TRANSITION FROM S22 TO S16
:= NOT as
END_TRANSITION

STEP S4_1:
m:= TRUE

END_STEP
```

References

[1] Robert W Lewis. *Programming industrial control systems using IEC 1131-3*. Number 50. Iet, 1998.

[2] Tadanao ZANMA, Tatsuya SUZUKI, Akio INABA, and Shigeru OKUMA. Transformation algorithm from ladder diagram to sfc using temporal logic. *IEEJ Transactions on Industry Applications*, 117(12):1471–1479, 1997.

[3] Albert Falcione and Bruce H Krogh. Design recovery for relay ladder logic. *IEEE Control Systems*, 13(2):90–98, 1993.

[4] Graphviz. About | graphviz - graph visualization software. URL: http://www.graphviz.org/About.php.

[5] Beremiz. Homepage of beremiz. URL: http://www.beremiz.org/.

[6] Sajid Iqbal and Ahsan Wasim. Programmable logic controllers (plcs): Workhorse of industrial automation. *New Horizons - Journal of The Institution of Electrical & Electronics Engineers Pakistan*, 68-69:27–31, 2010.

[7] MengChu Zhou and Edward Twiss. Design of industrial automated systems via relay ladder logic programming and petri nets. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 28(1):137–150, 1998.

[8] Programmable controllers - part 3: Programming languages. Standard, International Electrotechnical Comission, Geneva, CH, January 2003.

[9] Michael Tiegelkamp and Karl-Heinz John. Iec 61131-3: Programming industrial automation systems, 1995.

[10] PLCopen. Plcopen for efficiency in automation. URL: http://www.plcopen.org/.

[11] PLCopen. Plcopen motion control. URL: http://www.plcopen.org/pages/tc2_motion_control/index.htm.

[12] PLCopen. Plcopen safety. URL: http://www.plcopen.org/pages/tc5_safety/index.htm.

[13] PLCopen. Communication. URL: http://www.plcopen.org/pages/tc4_communication/index.htm.

[14] PLCopen. Communication. URL: http://www.plcopen.org/pages/tc6_xml/xml_intro/index.htm.

[15] PLCopen. Plcopen training. URL: http://www.plcopen.org/pages/pc2_training/index.htm.

[16] Beremiz. Beremiz documentation. URL: http://www.beremiz.org/doc.

[17] W3Schools. Xml tutorial, 2017. URL: http://www.w3schools.com/xml/default.asp.

[18] Xml formats for iec 61131-3. Technical paper, PLCopen, May 2009.

[19] RK Vimal Nandhan and N Ramesh Babu. Understanding of logic in ladder program with its transformation into sequential graph using state space-based approach. *International Journal of Mechatronics and Manufacturing Systems*, 6(2):159–182, 2013.

[20] Emden Gansner, Eleftherios Koutsofios, and Stephen North. Drawing graphs with dot - dot user's manual. Technical paper, 2006. URL: http://www.graphviz.org/Documentation/dotguide.pdf.